**Université de Montréal**

# Leveraging Deep Reinforcement Learning in the Smart Grid Environment.

par

**Ysaël Desage**

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Discipline

2020-05-21

# Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

**Leveraging Deep Reinforcement Learning in the Smart Grid Environment.**

présenté par

## Ysaël Desage

a été évalué par un jury composé des personnes suivantes :

*Emma Frejinger*
(président-rapporteur)

*Fabian Bastin*
(directeur de recherche)

*François Bouffard*
(codirecteur)

*Yoshua Bengio*
(membre du jury)

# Résumé

**Mots-clés**: Bâtiments, Apprentissage Profond, Apprentissage par Renforcement, Consommation Énergétique, Contrôle, Optimisation, Appels de Puissance, Réseaux Intelligents.

L'apprentissage statistique moderne démontre des résultats impressionnants, où les ordinateurs viennent à atteindre ou même à excéder les standards humains dans certaines applications telles que la vision par ordinateur ou les jeux de stratégie. Pourtant, malgré ces avancées, force est de constater que les applications fiables en déploiement en sont encore à leur état embryonnaire en comparaison aux opportunités qu'elles pourraient apporter.

C'est dans cette perspective, avec une emphase mise sur la théorie de décision séquentielle et sur les recherches récentes en apprentissage automatique, que nous démontrons l'application efficace de ces méthodes sur des cas liés au réseau électrique et à l'optimisation de ses acteurs. Nous considérons ainsi des instances impliquant des unités d'emmagasinement énergétique ou des voitures électriques, jusqu'aux contrôles thermiques des bâtiments intelligents. Nous concluons finalement en introduisant une nouvelle approche hybride qui combine les performances modernes de l'apprentissage profond et de l'apprentissage par renforcement au cadre d'application éprouvé de la recherche opérationnelle classique, dans le but de faciliter l'intégration de nouvelles méthodes d'apprentissage statistique sur différentes applications concrètes.

# Abstract

**Keywords**: Buildings, Deep Learning, Deep Reinforcement Learning, Energy Consumption, Optimal Control, Optimization, Power Consumption, Smart Grid.

While modern statistical learning is achieving impressive results, as computers start exceeding human baselines in some applications like computer vision, or even beating professional human players at strategy games without any prior knowledge, reliable deployed applications are still in their infancy compared to what these new opportunities could fathom.

In this perspective, with a keen focus on sequential decision theory and recent statistical learning research, we demonstrate efficient application of such methods on instances involving the energy grid and the optimization of its actors, from energy storage and electric cars to smart buildings and thermal controls. We conclude by introducing a new hybrid approach combining the modern performance of deep learning and reinforcement learning with the proven application framework of operations research, in the objective of facilitating seamlessly the integration of new statistical learning-oriented methodologies in concrete applications.

# Contents

# List of figures

# Liste des sigles et des abréviations

| | |
|---|---|
| A2C | Advantage Actor-Critic |
| A3C | Asynchronous Actor-Critic |
| AHU | Air-Handling Unit |
| ANN | Artificial Neural Network |
| APV-MCTS | Asynchronous Policy and Value Monte Carlo Tree Search |
| BPTT | Back-Propagation Through Time |
| BRNN | Bidirectional Recurrent Neural Network |
| CI | Confidence Interval |
| CLT | Central Limit Theorem |
| CNN | Convolutional Neural Network |
| CV | Cross-Validation |
| DP | Dynamic Programming |
| DDP | Deterministic Dynamic Program |
| DDPG | Deep Deterministic Policy Gradient |
| DQN | Deep Q-Network |
| DRL | Deep Reinforcement Learning |
| DRQN | Deep Recurrent Q-Network |
| ELU | Exponential Linear Unit |
| FTCS | Forward-in-Time-Centered-in-Space |
| GA | Global Adjustment |
| GRU | Gated Recurrent Unit |
| GPI | Generalized Policy Iteration |
| GUI | Graphical User Interface |
| HD | History dependent and Deterministic (policy) |
| HOEP | Hourly Ontario Energy Price |
| HR | History dependent and Randomized (policy) |
| HVAC | Heat Ventilation and Air-Conditioning |
| ICI | Industrial Conservative Initiative |
| KL | Kullback-Leibler (divergence) |

| | |
|---|---|
| KPI | Key Performance Indicators |
| LRCN | Long-term Recurrent Convolutional Networks |
| LSTM | Long Short-Term Memory |
| MAE | Mean Absolute Error |
| MAPE | Mean Absolute Percentage Error |
| MCNN | Multi-Scale Convolutional Neural Network |
| MCTS | Monte Carlo Tree Search |
| MD | Markovian and Deterministic (policy) |
| MDP | Markov Decision Process |
| ML | Machine Learning |
| MLP | Multi-Layer Perceptron |
| MPC | Model Predictive Control |
| MR | Markovian and Randomized (policy) |
| MRP | Markov Reward Process |
| MSE | Mean Squared Error |
| NLP | Natural Language Processing |
| OR | Operations Research |
| PDE | Partial Differential Equation |
| PDF | Probability Distribution Function |
| PMF | Probability Mass Function |
| POC | Proof of Concept |
| PPO | Proximal Policy Optimization |
| RE | Relative Error |
| ReLU | Rectified Linear Unit |
| RL | Reinforcement Learning |
| RMSE | Root Mean Squared Error |
| RNN | Recurrent Neural Network |
| RTU | Roof Top Unit |
| SAC | Soft Actor-Critic |
| Seq2Seq | Sequence to Sequence |
| SGD | Stochastic Gradient Descent |
| TRPO | Trust Region Policy Optimization |
| UAT | Universal Approximation Theorem |
| VRF | Variance Reduction Factor |
| VRT | Variance Reduction Technique |

# Remerciements

À vous tous, qui avez de près ou de loin croisé mon chemin ces dernières années, et influencé mon parcours pour me permettre d'en être là aujourd'hui - merci. Merci à mon entourage professionnel, académique, amical et familial de m'avoir accompagné, d'avoir cru en moi, et d'avoir été des ressources inestimables d'inspiration, de partage et de support dans ces temps rocambolesques où je me suis découvert un nouvel horizon de vie et une nouvelle passion.

Une reconnaissance particulière pour Françoise Desage, Richard Boudreault, Paul Charbonneau et Jean-Simon Venne, qui ont été des facteurs décisifs dans cette épopée, et qui ont grandement influencé l'homme, le scientifique et le chercheur que j'aspire à devenir aujourd'hui.

Finalement, des remerciements affirmés à Fabian Bastin et à François Bouffard, mes deux superviseurs de maîtrise. Tous deux se sont avérés des mentors exceptionnels, mais également des vecteurs d'opportunité, des recours de support en temps nécessaires, et des sources intarissables de judicieux conseils. Merci à eux d'avoir enduré mon entêtement parfois soutenu, de m'avoir laissé faire mes apprentissages nécessaires, et de toujours avoir été là à toute heure du jour et de la nuit pour m'aider dans mon cheminement. Cette aventure avec eux n'est pour moi que le début de grands projets partagés, avec bien entendu, le partage de nombreuse futures discussions riches et inusitées.

# Introduction

A tear rolled down the man's cheek as he stared blankly at the screen. He had lost his entire investment and years worth of trading, as the Daw Jones lived its biggest day-drop in history. A few hundred kilometers away, numbers just came out: hundreds of people died during the night. How can governments improve the actual COVID-19 virus mitigation strategy to save a maximum of lives while minimizing the impact on everyone's financial and mental stability? Just like two martial arts athletes facing each other intensively, wondering which one will strike first knowing that it may compromise their defence and make them momentarily vulnerable... yet if executed well, end the fight in a glorious outcome; all of these scenarios are examples of *decision making*. Decision making under *uncertainty*, which can either end up in a very expensive outcome if poor choices are made, or in other person's perspective, into a sizeable opportunity for great accomplishments.

In the same decade, while still far from general artificial intelligence, computer cognition is reaching impressive levels exceeding human baselines in some applications like computer vision, or even beating professional human talents at video and strategy games without any prior knowledge. Major breakthroughs and state-of-the-art results are now detained by computers in several fields. The main reason: statistical learning. The art of *learning* from experience. As this area of research and its neighbouring communities live an explosion of popularity in the recent years, with research growing exponentially and new knowledge being introduced on a daily basis, robust deployed applications are still in their infancy of what these new opportunities could bring.

While far from exhaustive, through this work we start by providing the reader with an overview of sequential decision making theory along with some recent improvements in statistical learning. We then proceed to demonstrate the application of recent developments in deep supervised learning and deep reinforcement learning on real applied instances involving the energy grid and the optimization of its actors, from energy storage and electric cars to smart building and thermal controls. We conclude by introducing a new hybrid approach combining the modern performance of statistical learning with the proven

application of classical operations research. We hope that such solution will facilitate the integration of new statistical learning-oriented methodologies, while blending smoothly and rigorously with other existing and future optimization solutions of different natures.

The first part of this document (chapter 1 to 3) introduces the formal mathematical framework of sequential decision making, along with its inherent characteristics. In the second part (chapter 4-6), we cover the fundamentals of statistical learning theory, with a very keen focus on deep learning and deep reinforcement learning, along with their modern approaches and methodologies. Finally, the third part describes the applications and main novel contributions of this thesis, building on the subjects introduced in the first six chapters. A reader familiar with either or both of control and statistical learning theories can skip directly to any part of this document without any loss of continuity.

# Part 1

# SEQUENTIAL DECISION FRAMEWORK

# Chapter 1

# Model Formulation

The modern study of stochastic sequential decision problems began with work on sequential statistical problems during the Second World War, and has seen thorough improvements and implementations through the ages and applications in our modern society. The term *dynamic programming* was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he refined this to the modern meaning, referring specifically to nesting smaller decision problems inside larger decisions [**20**]. Nowadays, research in this field inhabits a new efflorescence, considering modern computing resources and the imminent rise in popularity of Artificial Intelligence.

Mainly synthesized from Pierre-Luc Bacon's *Excursions in Reinforcement Learning* course, the *Dynamic Programming and Optimal Control (Vol I)* [**11**], *Dynamic Programming and Optimal Control (Vol II) - Approximate Dynamic Programming* [**9**] and *Reinforcement Learning and Optimal Control* [**12**] books from Dimitri P. Bertsekas, the *Markov Decision Processes* [**56**] book by Martin L. Puterman, and the *Reinforcement Learning* [**72**] book by Richard S. Sutton and Andrew G. Barto, we dive directly in the matter and start by considering the very broad and general problem in which a decision maker must choose a sequence of actions so as to maximize a given performance measure. These actions can be conceptualized as possible ways of influencing its environment over a series of interactions through time. Since the environment we model is ongoing, the state of the system prior to tomorrow's decision depends on today's decision. Consequently, decisions must not be made myopically, but must take account of the opportunities and costs (or rewards) associated with future possibilities.

## 1.1. Problem Definition

Uncertainty and stochasticity can arise from many sources - nearly all activities imply some ability to reason in the presence of uncertainty. In fact, beyond mathematical statements that are true by definition, it is difficult to think of any proposition that is absolutely true or any event that is absolutely guaranteed to occur. More specifically speaking, uncertainty arises from three possible sources [**51**]:

(1) Inherent stochasticity in the nature of the system - for example, the fundamental nature of quantum mechanics, which describe the dynamics of subatomic particles, has been proven to be probabilistic.
(2) Incomplete observability - Even deterministic systems can appear stochastic when the observer does not have access to all the variables that drive the behavior of the system.
(3) Incomplete modeling - When the model considered discards some of the information we have, usually for simplicity or efficiency reasons, the discarded information results in uncertainty in the prediction.

Mainly concerned by the last two sources (we shall leave the realm of the infinitely small out for all intents and purposes), we thus consider our *environment*[1], denoted $E$, as a probabilistic system. At a specified point in time, the *decision maker* or *agent* [2], denoted $a$, observes the state of this system. Based on this observation, the decision maker is faced with the problem (or some might say the opportunity) to choose an action (or a sequence of actions) which influences the system to perform optimally with respect to some predetermined performance criterion.

The decision maker can interact with the environment at different time instants, denoted $t$, called *decision epochs*, *decision steps* or just *steps*. Furthermore, the interval between two decision epochs in discrete time is referred to as *stages* or *periods*. The set of decision epochs can be either discrete, spaced out at irregular time intervals, or occurring continuously in time.
We refer to the set of decision epochs as $\mathcal{T}$, which leads to the classification:

- Discrete time, finite horizon: $\mathcal{T} := \{0,1,\ldots,T-1\}$ for $T < \infty$.
- Discrete time, infinite horizon: $\mathcal{T} := \{0,1,\ldots\}$.
- Continuous time, finite horizon: $\mathcal{T} := [0,1,\ldots,T-1]$.

[1] also referred to as *plant* in some literature.
[2] sometimes called *controller*.

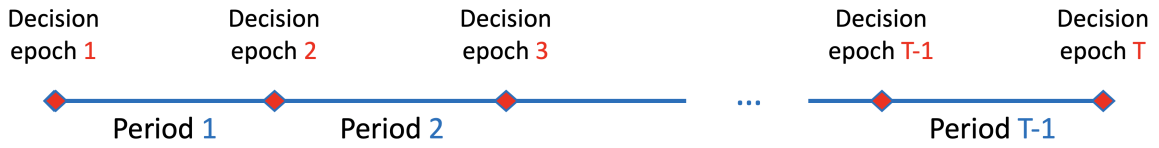


**Fig. 1.1.** Diagram of the interactions between the agent and the environment in the sequential decision-making framework.

- Continuous time, infinite horizon: $\mathcal{T} := [0, \infty)$.

In finite horizon problems, we typically assume that the decision maker is unable to take a decision in the last decision epoch since the outcome of this decision would have to be observed beyond the horizon.

### 1.1.1. State and Action Sets

At each decision epoch, the probabilistic system occupies a given *state*, represented by $s$, belonging to a set of states $\mathcal{S}$[3]. In every state, we assume that the decision maker can influence the underlying probabilistic system via an *action* or *control*, denoted $u$, from a set of allowable actions $\mathcal{U}_s$ which depend on state $s$. It is very common in the reinforcement learning literature to assume that $\mathcal{U}_s = \mathcal{U}$ for all $s \in \mathcal{S}$, where $\mathcal{U} := \cup_{s\in\mathcal{S}}\mathcal{U}$.

The sets $\mathcal{S}$ and $\mathcal{U}_s$ may each be either:

(1) arbitrary finite sets,
(2) arbitrary countably infinite sets,
(3) compact subsets of finite dimensional Euclidian space, or
(4) non-empty Borel subsets of complete, separable metric spaces.

In nondiscrete settings, many subtle mathematical issues arise which, while interesting, detract from the main purpose of the work presented. We thus assume, from now on, that $\mathcal{S}$ and $\mathcal{U}_s$ are *discrete* (finite and countably infinite) unless explicitly noted [4].

Actions may be chosen either randomly or deterministically. We denote by $\mathcal{P}(\mathcal{U}_s)$ and $\mathcal{P}(\mathcal{U})$ the collection of probability distributions on (Borel) subsets of $\mathcal{U}_s$ and $\mathcal{U}$, respectively. Choosing actions randomly means selecting a probability distribution $q(\bullet) \in \mathcal{P}(\mathcal{U}_s)$, in which

[3]We assume, in what we believe a reasonable hypothesis, the same state space at every decision epoch.
[4]This statement is unconditionally true for $\mathcal{S}$ in our case, but not always for $\mathcal{U}$.

case action $u$ is selected with probability $q(u)$. Degenerate probability distributions correspond to deterministic action choices.

### 1.1.2. System Evolution

As a result of choosing action $u \in \mathcal{U}_s$ in state $s$ at decision epoch $t$,

(1) The decision maker receives a reward (or cost), $r_t(s,u), r_t : \mathcal{S} \times \mathcal{U} \to \mathbb{R}$ and
(2) the system state at the next decision epoch is determined by the probability distribution $p_t(\bullet|s,u), p_t : \mathcal{S} \times \mathcal{U} \to [0,1]$.

Let the real-valued function $r_t(s,u)$ defined for $s \in \mathcal{S}$ and $u \in \mathcal{U}_s$ denote the value at time $t$ of the reward received in period $t$, and be additive over time. When positive, $r_t(s,u)$ may be regarded as income, and when negative as cost. It is meant to have an *immediate* meaning, which contrasts with the notion of *return*. The underlying process by which this reward is accrued does not need to be specified in our model: the only required specificity is the possibility to compute this value as a function of the current state and action. Furthermore, in the finite horizon setting, no decision is made at decision epoch $T$. Consequently, the reward at this time point is only a function of the state. We denote it by $r_T(s)$ and sometimes refer to it as a *salvage value* or *scrap value*.

When the reward depends on the state of the system at the next decision epoch, we let $r_t(s,u,j)$ denote the value at time $t$ of the reward received when the state of the system at decision epoch $t$ is $s$, action $u \in \mathcal{U}_s$ is selected, and the system occupies state $j$ at decision epoch $t+1$. This setting can be reduced to the *standard* formulation by marginalizing out the next state:

$$r_t(s_t,u_t) = \sum_{j \in \mathcal{S}} r_t(s_t,u_t,j) p_t(j|s_t,u_t) \ . \tag{1.1.1}$$

The function $p_t(j|s,u)$, subject to the constraint $\sum_{j \in \mathcal{S}} p_t(j|s,u) = 1$, is called a *transition probability function*. It is important to note that many system transitions might occur in the time period between decision epoch $t$ and decision epoch $t+1$. Under most notions of optimality, all of the information is necessary to make a decision at time $t$ is summarized in $r_t(s,u)$ and $p_t(j|s,u)$; however, under some criteria we must use $r_t(s,u,j)$ instead of $r_t(s,u)$.

We define a *Markov decision process* (MDP) as the following collection of objects:

- a set of decision epochs,
- a set of system states,

- a set of available actions,
- a set of state and action dependent transition probabilities, and
- a set of state and action dependent immediate rewards or costs:

$$\{\mathcal{T},\mathcal{S},\mathcal{U}, p_t(\bullet|s,u),r_t(s,u)\} \ . \tag{1.1.2}$$

The qualifier Markov is used because the transition probability and reward functions depend on the past only through current state of the system and the action selected by the decision maker in that state. Furthermore, we say that the combination of an MDP with a performance measure describes a *Markov Decision Problem.*

### 1.1.3. Taking Decisions

The information regarding which action to choose at any given state and decision epoch is encoded into a *decision rule*, denoted $d$. Decision rules range in generality from deterministic Markovian to randomized history dependent, depending on how they incorporate past information and how they select actions. Deterministic Markovian decision rules are functions $d_t : \mathcal{S} \to \mathcal{U}_s$, which specify the action choice when the system occupies state $s$ at decision epoch $t$. For each $s \in \mathcal{S}$, $d_t(s) \in \mathcal{U}_s$. This decision rule is said to be *Markovian* (memoryless) because it depends on previous system states and actions only through the current state of the system, and deterministic because it chooses an action with certainty. We call a deterministic decision rule *history dependent* if it depends on the past history of the system as represented by the sequence of previous states and actions. That is, $d_t$ is a function of the history $h_t = (s_1,u_1,...,s_{t-1},u_{t-1},s_t)$. The set of all possible histories at time $t$ can be written recursively as $\mathcal{H}_t := \mathcal{H}_{t-1}\times\mathcal{U}\times\mathcal{S}$, where $\mathcal{H}_0 = \mathcal{S}$.

A *randomized* decision rule $d_t$ specifies a probability distribution $q_{d_t}(\bullet)$ on the set of actions. Randomized Markovian decision rules map the set of states into the set of probability distributions on the action space, that is $d_t : \mathcal{S} \to \mathcal{P}(\mathcal{U})$, and randomized history-dependent decision rules according to $d_t : H_t \to \mathcal{P}(\mathcal{U})$. When Markovian, $q_{d_t(s_t)}(\bullet) \in \mathcal{P}(\mathcal{U}_{s_t})$, and when history dependent, $q_{d_t(h_t)}(\bullet) \in \mathcal{U}_{s_t}$ for all $h_t \in \mathcal{H}_t$. A deterministic decision rule may be regarded as a special case of a randomized decision rule in which the probability distribution on the set of actions is degenerate, that is, $q_{d_t(s_t)}(u) = 1$ or $q_{d_t(h_t)}(u) = 1$ for some $u \in \mathcal{U}_s$.

Using this classification, we therefore have decision rules that are history dependent and randomized (HR), history dependent and deterministic (HD), Markovian and randomized (MR), or Markovian and deterministic (MD) depending on their degree of dependence on past information and on their method of action selection. We denote the set of decision

rules at time $t$ by $\mathcal{D}_t^K$, where $K$ designates a class of decision rules ($K$ = HR, HD, MR, MD); $\mathcal{D}_t^K$ is called a *decision rule set*.

The rewards and transition probabilities become functions on $\mathcal{S}$ or $\mathcal{H}_t$ after specifying decision rules. For $d_t \in D_t^{\mathrm{HD}}$, they equal $r_t(s,d_t(h_t))$ and $p_t(j|s,d_t(h_t))$ whenever $h_t = (h_{t-1},u_{t-1},s)$. If $d_t$ is a randomized Markov decision rule the *expected* reward satisfies

$$r_t(s,d_t(s)) = \sum_{u\in\mathcal{U}_s} r_t(s,u)q_{d_t(s)}(u) \ , \tag{1.1.3}$$

and the transition probability satisfies

$$(j|s,d_t(s)) = \sum_{u\in\mathcal{U}_s} p_t(j|s,u)q_{d_t(s)}(u) \ . \tag{1.1.4}$$

Analogous constructions apply to randomized history-dependent rules.

A decision rule only dictates the choice of action in a given state. A *policy*, *contingency plan*, *plan*, *control law*, or *strategy* specifies the decision rule to be used at all decision epoch. It provides the decision maker with a prescription for action selection under any possible future system state or history. A policy $\pi$ is a sequence of decision rules, i.e. $\pi = (d_0,d_1,...)$ where $d_t$ maps states of $s_t$ into controls $u_t = d_t(s_t)$. A *stationary policy* is a policy where the same decision rule (which is not to say the same action) is chosen in every decision epoch: $\pi := (d,d,...)$. The relationship between the various classes of policies is as follows:

$$\Pi^{\mathrm{SD}} \subset \Pi^{\mathrm{SR}} \subset \Pi^{\mathrm{MR}} \subset \Pi^{\mathrm{HR}} \tag{1.1.5}$$

$$\Pi^{\mathrm{SD}} \subset \Pi^{\mathrm{MD}} \subset \Pi^{\mathrm{MR}} \subset \Pi^{\mathrm{HR}} \tag{1.1.6}$$

$$\Pi^{\mathrm{SD}} \subset \Pi^{\mathrm{MD}} \subset \Pi^{\mathrm{HD}} \subset \Pi^{\mathrm{HR}} \tag{1.1.7}$$

where $\Pi$ represents the space of the specified policy class, and the letters S,D,R and H are respectively acronyms for stationary, deterministic, randomized and history-dependent. Consequently, randomized, history-dependent policies are most general and stationary deterministic policies are most specific. Stationary deterministic policies are appealing due to their simple structure. In the infinite horizon criteria, we will see that we can restrict our attention to such pure policies without any loss of optimality.

Finally, we also note the distiction between *open-loop* minimization, where we select all controls $u_0,...,u_{T-1}$ at once at time 0, and *closed-loop* minimization, where we select a policy $\{d_0,...,d_{T-1}\}$ that applies the control $d_t(s_t)$ at time $t$ with knowledge of the current state $s_t$. With close-loop policies, it is possible to achieve higher rewards, essentially by taking advantage of the extra information (the knowledge of the current state). The reduction in

cost may be called the *value of the information* and can be significant. If the information is not available, the controller cannot adapt appropriately to unexpected values of the state, and as a result the cost can be adversely affected.

### 1.1.4. Induced Stochastic Processes

The combination of a policy and an MDP induces a stochastic process whose sample space $\Omega$ is of the form:

$$\Omega = \mathcal{S} \times \mathcal{U} \times \mathcal{S} \times \mathcal{U} \times ... \times \mathcal{U} \times \mathcal{S} = \{\mathcal{S} \times \mathcal{U}\}^{T-1} \times \mathcal{S} \ , \tag{1.1.8}$$

and in an infinite horizon model, $\Omega = \{\mathcal{S} \times \mathcal{U}\}^{\infty}$. A typical element, or *sample path*, $\omega \in \Omega$ consists of a sequence of states and actions, that is:

$$\omega = (s_1,u_1,s_2,u_2,...,u_{T-1},s_T) \ \text{ or } \ \omega = (s_1,u_1,s_2,...) \ . \tag{1.1.9}$$

From a sample path, we define the random variables for the state and actions at time $t$, which we write $S_t(\omega) = s_t$ and $U_t(\omega) = u_t$, as well as the history of the process $H_t(\omega) := h_t(s_0,u_0,...,u_{t-1},s_t)$.

Let the probability distribution $P_1(\bullet)$ denote the *initial distribution* of the system state. In most applications, we assume degenerate $P_1(\bullet)$, that is, $P_1(s_1) = 1$ for some $s_1 \in \mathcal{S}$. A randomized history-dependent policy $\pi = (d_1,d_2,...,d_{T-1})$, $T \leq \infty$, induces a probability distribution $P^{\pi}$ through

$$P^{\pi}\{S_1 = s\} = P_1(s) \ , \tag{1.1.10}$$

$$P^{\pi}\{U_t = u | H_t = h_t\} = q_{d_t(h_t)}(u) \ , \tag{1.1.11}$$

$$P^{\pi}\{S_{t+1} = s | H_t = (h_{t-1},u_{t-1},s_t), U_t = u_t\} = p_t(s|s_t,u_t) \ , \tag{1.1.12}$$

so that probability of a sample path $\omega = (s_1,u_1,s_2,...,s_T)$ is given by

$$P^{\pi}(s_1,u_1,s_2,...,s_T) = P_1(s_1)q_{d_1(s_1)}(u_1)p_1(s_2|s_1,u_1)q_{d_2(h_2)}(u_2)... \tag{1.1.13}$$

$$q_{d_{T-1}(h_{T-1})}(u_{T-1})p_{T-1}(s_T|s_{T-1},u_{T-1}) \ . \tag{1.1.14}$$

for $\pi$ in $\Pi^{\text{HD}}$ or $\Pi^{\text{MD}}$, this simplifies to

$$P^{\pi}(s_1,u_1,s_2,...,s_T) = P_1(s_1)p_1(s_2|s_1,u_1)...p(s_T|s_{T-1},u_{T-1}) \ . \tag{1.1.15}$$

For computation, we require conditional probabilities of the process from $t$ onward, conditional on the history at time $t$. Under the discreteness assumptions, we compute these probabilities as follows:

$$P^{\pi}(u_1,s_{t+1},...,s_T|s_1,u_1,...,s_t) = \frac{P^{\pi}(s_1,u_1,...,s_T)}{P^{\pi}(s_1,u_1,...,s_t)} \ . \tag{1.1.16}$$

Few lines of development and algebra lead to

$$P^{\pi}(u_t,s_{t+1},...,s_T|s_1,u_1,...,s_t) = P^{\pi}(u_t,s_{t+1},...,s_T|s_t) \ , \tag{1.1.17}$$

so that the induced stochastic processes $\{S_t; t \in \mathcal{T}\}$ is a discrete Markov chain.

When $\pi$ is Markovian, we refer to the bivariate stochastic process $\{(S_t,r_t(S_t,U_t)); t \in \mathcal{T}\}$ as a *Markov Reward Process* (MRP) (a Markov chain together with a real-valued function defined on its state space). This process represents the sequence of system states and stream of rewards received by the decision maker when using policy $\pi$.

## 1.2. Deterministic Dynamic Programs

Deterministic dynamic programs (DDPs) represent an important and widely studied class of problems. They arise when the transition probability function is specified by a *transfer function* indicating exactly what will be the next state. The transition probability function can then be written as:

$$p_t(s_{t+1}|s_t,u_t) = \begin{cases} 1 & \text{if } \ s_{t+1} = f_t(s_t,u_t) \ , \\ 0 & \text{if } \ s_{t+1} \neq f_t(s_t,u_t) \ , \end{cases} \tag{1.2.1}$$

and where the rewards are given by $r_t(s,u)$. When the total reward is used to compare policies, every DDP with finite $\mathcal{S}, \mathcal{U}$ and $\mathcal{T}$ is equivalent to a shortest or longest path problem.

A *finite directed graph* consists of a set of *nodes* and directed *arcs* or *edges*. We define a *path* as a sequence of arcs that connects one node to another node. We call such a graph *acyclic* whenever there are no paths which begin and end at the same node. The first (initial) node is called the *origin*, while the last (final) node is called the *destination*. Finally, each node and edge can have attributes and or properties associated to it, such as a distance, a cost, a time etc.

Formulation of a shortest path problem as a deterministic dynamic program involves identifying nodes with states, arcs with actions, transfer functions with transition probabilities and values with rewards. However, direct identification of decision epochs or stages is

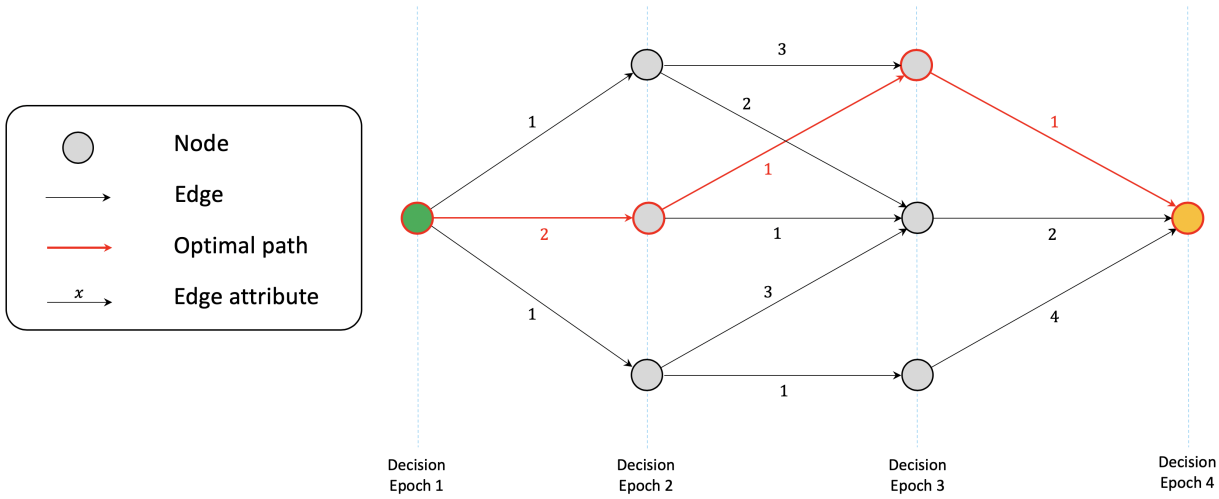


**Fig. 1.2.** Deterministic dynamic program expressed as a shortest route problem. The origin node is colored in green, the terminal node in orange, and the red path represents the optimal (shortest) path to minimize the overall cost attribute up to decision epoch $T = 3$.

not always possible and not necessary for solution. Looking at figure 1.2, at each decision epoch, there is a node corresponding to each state. Arcs originate at each state, one for each action, and end at the next stage at the node determined by the corresponding transfer function. Arc lengths give rewards (or costs). We also add a "dummy" origin (O) and destination (D). Arc lengths from nodes at stage $T$ to the destination give terminal rewards $r_T(s)$. If a solution is sought for a particular initial state $s'$ and the problem is one of maximization, then we give the arc from the origin to $s'$ a large positive lengths, say $L$, and set length of all other arcs starting from the origin equal to zero.

Several specialized algorithms and methods can be used to solve the shortest path problem induced by the MDP. We do not develop the full detail of these algorithms explicitly, but an interested reader can easily find all the necessary material for *label correcting* methods (like Dijkstra), the *Branch and Bound* method and many more, to only name those few most popular. As we will see, the backward induction algorithm presented in the next section can also be used to solve our shortest path problem. It is however noteworthy to mention that depth-first graph exploration is usually favored in DDPs.

## 1.3. Controlled Discrete-Time Dynamic Systems

When decisions are made continuously, the sequential decision problems are best analyzed using control theory methods based on dynamic system equations. Under this perspective[5],

[5]Discrete-time optimal control originates from the calculus of variations and its application for continuous-time systems.

we define a *state equation* (also called *system equation* or *law of motion*) which describes how the system evolves over time under some exogenous source of disturbances:

$$s_{t+1} = f_t(s_t,u_t,w_t), \quad t = 0, 1, ..., T-1 \ , \tag{1.3.1}$$

where

- $t$ indexes discrete time,
- $s_t$ is the state of the system and summarizes past information that is relevant for future optimization,
- $u_t$ is the control or decision variable to be selected at time $t$,
- $w_t$ is a random parameter (also called *disturbance* or *noise* depending on the context),
- $T$ is the horizon or number of times control is applied,
- $f_t$ is a function that describes the system and in particular the mechanism by which the state is updated.

We formulate the model so that transitions which occur between decision epochs do not influence the decision maker. This type of state transition can alternatively be described in terms of the discrete-time system equation

$$s_{t+1} = w_t \ , \tag{1.3.2}$$

where the probability distribution of the random parameter $w_t$ is

$$P\{w_t = j | s_t = i, u_t = u\} = p_{ij}(u,t) \ , \tag{1.3.3}$$

Conversely, given a discrete-state system in the form

$$s_{t+1} = f_t(s_t,u_t,w_t) \ , \tag{1.3.4}$$

together with the probability distribution $P_t\{w_t|s_t, u_t\}$ of $w_t$, we can provide an equivalent transition probability description. The corresponding transition probabilities are given by

$$p_{ij}(u,t) = P_t\{W_t(i,u,j)|s_t = i, u_t = u\} \ , \tag{1.3.5}$$

where $W(i,u,j)$ is the set

$$W_t(i,u,j) = \{w | j = f_t(i,u,w)\} \ . \tag{1.3.6}$$

Thus a discrete-state system can equivalently be described in terms of a difference equation or in terms of transition probabilities. Depending on the given problem, it may be notationally or mathematically more convenient to use one description over the other.

# Chapter 2

# Finite Horizon Markov Decision Processes

This chapter focuses on finite-horizon discrete-time MDPs, i.e. problems in which there is a clearly defined final decision epoch $T < \infty$. This type of problem is very typical of what is referred to as *episodic* experiences in reinforcement learning, while its *continuous* counterpart is developed in the next chapter with the infinite-horizon setting.

## 2.1. Optimality Criteria

### 2.1.1. Some Preliminaries

Let $\pi = (d_1,d_2,...,d_{T-1})$ denote a randomized history-dependent policy. Each $\pi \in \Pi^{\text{HR}}$ generates a probability distribution $P^\pi(\bullet)$ on (Borel subsets of) $\mathcal{H}_T$, the set of histories (sample paths) up to time $T$. For each realization of the history $h_T = (s_1,u_1,s_2,...,s_T)$ there corresponds a sequence of rewards $\{r_1(s_1,u_1),...,r_{T-1}(s_{T-1},u_{T-1}),r_T(s_T)\}$. Let $R_t \equiv r_t(S_t,U_t)$ denote the random reward received in period $t < T$, $R_T \equiv r_T(S_T)$ denote the terminal reward, $R \equiv (R_1,R_2,...,R_T)$ denote a random sequence of rewards, and $\mathcal{R}$ the set of all possible reward sequences. A policy $\pi$ induces a probability distribution $P^\pi_{\mathcal{R}}(\bullet)$ on (Borel subsets of) $\mathcal{R}$:

$$P^\pi_{\mathcal{R}}(p_1,...,p_T) \equiv P\Big[\{(s_1,u_1,...,s_N) : r_1(s_1,u_1),...,r_{T-1}(s_{T-1},u_{T-1}),r_T(s_T)\} = (p_1,...,p_T)\Big]. \tag{2.1.1}$$

In this definition, $P^\pi[\bullet]$ denotes the probability distribution on the set of all realizations of the Markov decision process under $\pi$. For a deterministic history dependent $\pi$, we have $U_t = d_t(h_t)$, while for $\pi \in \Pi^{\text{MD}}$ $U_t = d_t(S_t)$. We compare policies on the basis of the decision maker's preference for different realizations of $R$ and the probability that each occurs. We now discuss approaches for such comparisons based on stochastic orderings, expected utility, and other criteria.

### 2.1.2. The Expected Total Reward Criterion

Let $v_T^\pi$ represent the expected total reward over the decision making horizon if policy $\pi$ is used and the system is in state $s$ at the first decision epoch. For $\pi \in \Pi^{\text{HR}}$, it is defined by

$$v_T^\pi(s) := \mathbb{E}\left\{ \sum_{t=0}^{T-1} r_t(S_t, U_t) + r_T(S_T) \,\middle|\, S_0 = s \right\} \ , \tag{2.1.2}$$

and where the expectation is taken over the distribution of sample paths induced by the policy $\pi$ in the given MDP. When using deterministic policies, the remaining source of randomness is due to the transition probability function, and the expected total reward can be written as:

$$v_T^\pi(s) := \mathbb{E}\left\{ \sum_{t=0}^{T-1} r_t(S_t, d_t(S_t)) + r_T(S_T) \,\middle|\, S_0 = s \right\} \ . \tag{2.1.3}$$

To account for the time value of rewards, we often introduce a discount factor $\gamma$, $0 \leq \gamma \leq 1$, which measures the value at time $n$ of a one unit reward received at time $n+1$. A one-unit reward received $t$ periods in the future has present value $\gamma^t$. When $\gamma = 1$, we assign equal importance to every reward irrespective of their occurrence in time while setting $\gamma = 0$ ignores all terms but the first one (myopically). For $\pi \in \Pi^{\text{HR}}$, the *expected total discounted reward* is

$$v_{T,\gamma}^\pi(s) := \mathbb{E}\left\{ \sum_{t=0}^{T-1} \gamma^t r_t(S_t, U_t) + \gamma^T r_T(S_T) \,\middle|\, S_0 = s \right\} \ , \tag{2.1.4}$$

Taking the discount factor into account does not effect any theoretical results or algorithms in the finite-horizon case but might effect the decision maker's preference for policies. As we will see, the discount factor plays a key analytic role in the infinite-horizon models setting.

### 2.1.3. Optimal Policies

Markov decision process theory and algorithms for finite-horizon models primarily concern finding an optimal policy which maximizes the total expected reward, or its discounted variant. This optimal policy has the property that:

$$v_T^{\pi^*}(s) \geq v_T^\pi(s), \ s \in \mathcal{S} \ , \tag{2.1.5}$$

for all $\pi \in \Pi^{\text{HR}}$. We refer to such a policy as an *optimal policy*. If an optimal policy cannot be obtained, then we seek an *$\epsilon$-optimal policy*, that is, for an $\epsilon > 0$, a policy $\pi_\epsilon^*$ with the property that

$$v_T^{\pi^*}(s) + \epsilon \geq v_T^\pi(s), \ s \in \mathcal{S} \ . \tag{2.1.6}$$

We seek to characterize the *value* of the Markov decision problem, $v_T^*$, defined by

$$v_T^*(s) \equiv \sup_{\pi \in \Pi^{\mathrm{HR}}} v_T^\pi(s), \ s \in \mathcal{S} \ , \tag{2.1.7}$$

and when the supremum is attained, by

$$v_T^*(s) = \max_{\pi \in \Pi^{\mathrm{HR}}} v_T^\pi(s), \ s \in \mathcal{S} \ . \tag{2.1.8}$$

The expected total reward of an optimal policy $\pi^*$ satisfies

$$v_T^{\pi^*}(s) = v_T^*(s), \ \forall s \in \mathcal{S} \ . \tag{2.1.9}$$

In practice, all that might be required is an optimal policy for some specified initial state. Alternatively, we might seek a policy prior to knowing the initial state. In this case, we seek a $\pi \in \Pi^{\mathrm{HR}}$ which maximizes $\sum_{s \in \mathcal{S}} v_T^\pi(s) P_1\{S_1 = s\}$. Clearly we may find such a policy by maximizing $v_T^\pi(s)$ for each $s$ for which $P_1\{S_1 = s\} > 0$. We use the expression *Markov decision problem* for a Markov decision process together with an optimality criteria.

## 2.2. Finite-Horizon Policy Evaluation

The very notion of optimal policies relies on the ability to compare policies in terms of their expected total reward (a partial order). Hence, a prerequisite for this task is to be able to at least find out what is the expected total reward associated with any given policy: the policy evaluation problem. A basic algorithm can be obtained by backward induction, starting from $t = T$ and $v_T = r_T$ down to $t = 0$:

$$v_t^\pi(s) = r_t(s_t, d_t(h_t)) + \sum_{s_{t+1} \in \mathcal{S}} p_t(s_{t+1} | s_t, d_t(h_t)) v_{t+1}^\pi((h_t, d_t(h_t), s_{t+1})) \ , \tag{2.2.1}$$

where $(h_t, u_t, s_{t+1}))$ is the concatenation of the history up to time $t$ (and ending with $s_t$) with $u_t$ and the next state. In the case of Markovian policies, we have:

$$v_t^\pi(s) = r_t(s_t, d_t(s_t)) + \sum_{s_{t+1} \in \mathcal{S}} p_t(s_{t+1} | s_t, d_t(s_t)) v_{t+1}^\pi(s_{t+1}) \ . \tag{2.2.2}$$

## 2.3. Bellman Optimality Equations

In this section we introduce one of the most fundamental results in sequential decision theory, the *Bellman optimality equations*[1], and investigate their inherent properties. We show that solutions of these equations correspond to optimal value functions and that they also provide a basis for determining optimal policies.

[1]Also sometimes referred to as *Bellman equations*, *optimality equations* or *function equations*.

Let

$$v_t^*(h_t) = \sup_{\pi \ \Pi^{\mathrm{HR}}} v_t^\pi(h_t) \ , \tag{2.3.1}$$

which denotes the supremum over all policies of the expected total reward from decision epoch $t$ onward when the history up to time $t$ is $h_t$. Since we know $h_t$, we only consider portions of policies from decision epoch $t$ onwards; that is, we require only the supremum over $(d_t,d_{t+1},...,d_{T-1}) \in \mathcal{D}_t^{\mathrm{HR}} \times \mathcal{D}_{t+1}^{\mathrm{HR}} \times ... \times \mathcal{D}_{T-1}^{\mathrm{HR}}$. When minimizing costs instead of maximizing rewards, we sometimes refer to $v_t^*$ as a *cost-to-go* function.

The optimality equations are given by

$$v_t^*(h_t) = \sup_{u \in \mathcal{U}_{s_t}} \left\{ r_t(s_t,u) + \sum_{j \in \mathcal{S}} p_t(j|s_t,u) v_{t+1}(h_t,u,j) \right\} \ , \tag{2.3.2}$$

for $t = 1, ..., T-1$ and $h_t = (h_{t-1},u_{t-1},s) \in \mathcal{H}_t$. For $t = T$, we add the boundary condition

$$v_T(h_T) = r_T(s_T) \ , \tag{2.3.3}$$

for $h_T = (h_{T-1},u_{T-1},s_T) \in \mathcal{H}_T$. These equations become *policy evaluation* equations when we replace the supremum over all actions in state $s_t$, by the action corresponding to a specified policy, or equivalently, when $\mathcal{U}_s$ is a singleton for each $s \in \mathcal{S}$. It is important to note that while they resemble the policy evaluation equations, they differ in the fact that they are inherently nonlinear.

The operation sup is implemented by evaluating the quantity in brackets for each $u \in \mathcal{U}$, and then choosing the supremum over all of these values. If the supremum is attained, for example, when each $\mathcal{U}_{s_t}$ is finite, it can be replaced by "max" so the optimality equations become

$$v_t^*(h_t) = \max_{u \in \mathcal{U}_{s_t}} \left\{ r_t(s_t,u) + \sum_{j \in \mathcal{S}} p_t(j|s_t,u) v_{t+1}(h_t,u,j) \right\} \ . \tag{2.3.4}$$

A solution to this system of equations and boundary condition is a sequence of functions $v_t : H_t \to R, t = 1,...,T$, with the property that $v_T$ satisfies the boundary condition $v_T(h_T) = r_T(s_T)$, $v_{T-1}$ satisfies the $(T-1)$th equation with $v_T$ substituted into the right-hand side of the $(T-1)$th equation, and so forth.

Our development so far has been general and we considered searching of the largest possible class of policies: that of history dependent randomized policies. This is cumbersome because the space of such policies grows exponentially as a function of the horizon. A central result in the theory of MDPs is that a deterministic Markov policy which is optimal

not only exists but can also be found by backward induction [2]. This result relies first on the fact that optimal value function depends on the history only through the current state. Intuitively, the inductive argument behind this clearly follows from the fact that the Bellman optimality equations maintains the history *uselessly* as all we need to compute the value in the next decision epoch is the current state itself. Combining this fact with the theorem that establishes the existence of an $\epsilon$-optimal deterministic history dependent policy, entails that

$$v_T^\star(s) = \max_{\pi \in \Pi^{\text{HR}}} v_T^\pi(s) = \max_{\pi \in \Pi^{\text{MD}}} v_T^\pi(s) \ . \tag{2.3.5}$$

This result simplifies the original optimality equations significantly, which now look like

$$v_t^\star(s_t) = \max_{u_t \in \mathcal{U}_{s_t}} r_t(s_t,u_t) + \sum_{s_{t+1}} p_t(s_{t+1}|s_t,u_t) v_{t+1}^\star(s_{t+1}) \ , \tag{2.3.6}$$

starting with $v_T^\star = r_T$.

# 2.4. Backward Induction

We conclude this chapter with an important algorithm directly derived from the recursive nature of the Bellman optimality equations, and usually known as the *backward induction* or *dynamic programming* algorithm. The term backward induction and dynamic programming are synonymous, although the latter often refers to all results and methods for sequential decision processes.

Backward induction provides an efficient method for solving finite-horizon discrete-time MDPs presenting an acyclic transition graph. For stochastic problems, enumeration and evaluation of *all* policies from the terminal state is the only alternative, but forward induction and reaching methods provide alternative solution methods for deterministic systems. Bellow we define the backward induction algorithm and shows how to use it to find optimal policies and value functions. The algorithm also generalizes policy evaluation.

## 2.4.1. The Backward Induction Algorithm

**1.** Set $t = T$ and

$$v_T^\star(s_T) = r_T(s_T) \ \forall \ s_T \in \mathcal{S} \tag{2.4.1}$$

**2.** Substitute $t-1$ for $t$ and compute $u_t^*(s_t) = \sup_{\pi \in \Pi} u_t^\pi$, for each $s_t \in \mathcal{S}$ by

[2] We omit the proof of this result for concision, but the complete development can be found both in Bertsekas' and Puterman's references.

$$v_t^\star(s_t) = \max_{u \in \mathcal{U}_{s_t}} \Big\{ r_t(s_t,u) + \sum_{j \in \mathcal{S}} p_t(j|s_t,u) v_{t+1}^*(j)) \Big\}. \tag{2.4.2}$$

Set

$$u_t^\star(s_t) = \arg\max_{u \in \mathcal{U}_{s_t}} \Big\{ r_t(s_t,u) + \sum_{j \in \mathcal{S}} p_t(j|s_t,u) u_{t+1}^*(j)) \Big\}. \tag{2.4.3}$$

**3.** If $t = 1$, stop. Otherwise return to step 2.

# Chapter 3

# Infinite Horizon Markov Decision Processes

In this chapter, we develop infinite-horizon Markov decision processes with the expected total discounted reward optimality criterion. We briefly introduce total reward and average reward settings, but our main focus will concern the discounted version, as these models are the best understood and most popular of all infinite-horizon Markov decision problems. Results for these models provide a standard for the theory of models with other optimality criteria. Results for discounted models are noteworthy in that they hold regardless of the chain structure of Markov chains generated by stationary policies.

Throughout this whole chapter, we will consider the following assumptions:

(1) *Stationary rewards and transition probabilities* - $r(s,u)$ and $p(j|s,u)$ do not vary from decision epoch to decision epoch.
(2) *Bounded rewards* - $|r(s,u)| \leq M < \infty, \ \forall u \in \mathcal{U}_s$ and $s \in \mathcal{S}$.
(3) *Discounting* - future rewards are discounted according to a discount factor $\gamma \in [0,1)$.
(4) *Discrete state spaces* - $\mathcal{S}$ is finite or countable.

## 3.1. Infinite-Horizon Policy Evaluation

Just like we did in the previous chapter, in this section we develop recursions for the expected total discounted reward of a Markov policy $\pi$. These recursions provide the basis for the vast majority of algorithms in optimal control and reinforcement learning theories.

Infinite-horizon models require evaluation of infinite sequences of rewards at all states in $\mathcal{S}$. Consequently, we need some notions of convergence of functions on $\mathcal{S}$. Usually the term convergence is used in the *pointwise convergence* sense, that is, limits are defined separately for each $s \in \mathcal{S}$. In models with expected total discounted reward, we analyze convergence

of algorithms and series in terms of convergence in supremum norm.

We say that the limit of a series *exists* whenever the series has a unique limit point, even though it might be $+\infty$ or $-\infty$. This distinguishes a divergent series, that is, with limits of either $+\infty$ or $-\infty$, from a non-convergent series which has multiple limit points.

Stationary policies assume a particularly important role in infinite-horizon models. We use the notation $d^\infty = (d,d,...)$ to denote this policy. In a stationary infinite-horizon Markov decision process, each policy $\pi = (d_1,d_2,...)$ induces a bivariate discrete-time reward process; $\{(S_t,r(S_t,U_t));\ t = 1,2,...\}$. The first component $S_t$ represents the state of the system at time $t$ and the second component represents the reward received when using action $U_t$ in state $S_t$. The decision rule $d_t$ determines the action $U_t$ as follows:
For deterministic $d_t$,

$$U_t = d_t(S_t) \ \text{ for } d_t \in \mathcal{D}^{\mathrm{MD}} \ \text{ and } U_t = d_t(H_t) \ \text{ for } d_t \in \mathcal{D}^{\mathrm{HD}}, \tag{3.1.1}$$

where the random variable $H_t$ denotes the history up to time $t$. For randomized $d_t$,

$$P\{U_t = u\} = q_{d_t(S_t)}(u) \ \text{ for } d_t \in \mathcal{D}^{\mathrm{MR}} \tag{3.1.2}$$

and

$$P\{U_t = u\} = q_{d_t(H_t)}(u) \ \text{ for } d_t \in \mathcal{D}^{\mathrm{HR}} \tag{3.1.3}$$

For Markovian $\pi$, $\{(S_t,r(S_t,U_t));\ t = 1,2,...\}$ is a Markov reward process.

For each $s \in \mathcal{S}$ given any $\pi \in \Pi^{\mathrm{HR}}$, there exists a $\pi' \in \Pi^{\mathrm{MR}}$ with identical total discounted rewards. Considering this, we need not consider history-dependent policies, so that

$$v_\gamma^*(s) \equiv \sup_{\pi \in \Pi^{\mathrm{HR}}} v_\gamma^\pi(s) = \sup_{\pi \in \Pi^{\mathrm{MR}}} v_\gamma^\pi(s) \tag{3.1.4}$$

The following functions assign values to the reward streams generated by a *fixed* policy when the system starts in a fixed state.

**a.** The *expected total reward* of policy $\pi \in \Pi^{\mathrm{HR}}$, $v^\pi$ is defined to be

$$v_\pi(s) = \lim_{T\to\infty} \mathbb{E}\left[\sum_{t=0}^{T} r(S_t,U_t) \,\middle|\, S_0 = s\right] = \lim_{T\to\infty} v_T^\pi(s) \ . \tag{3.1.5}$$

When the limit exists and when the limit can be interchanged with the expectation (by Lebesgue's dominated convergence theorem for example), we can just write:

$$v_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} r(S_t, U_t) \,\middle|\, S_0 = s\right] \quad . \tag{3.1.6}$$

**b.** To ensure that the expectation is finite, we can use a discount factor $\gamma \in [0, 1)$ resulting in the *expected total discounted reward*:

$$v_{\pi,\gamma}(s) = \lim_{T\to\infty} \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r(S_t, U_t) \,\middle|\, S_0 = s\right] \quad . \tag{3.1.7}$$

The discount factor appears along the reward only from $t = 1$ because the realization of the reward random variable is only available upon entering the next state after taking the first decision. When the reward function is bounded and $\gamma \in [0,1)$, the limit exists and we write:

$$v_{\pi,\gamma}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(S_t, U_t) \,\middle|\, S_0 = s\right]. \tag{3.1.8}$$

Note that if the maximum possible reward is bounded by some constant $r_{\text{MAX}}$:

$$\max_{s\in\mathcal{S}} \max_{u\in\mathcal{U}} |r(s,u)| = r_{\text{MAX}} \quad , \tag{3.1.9}$$

then the return is at most $\frac{r_{\text{MAX}}}{1-\gamma}$ since we have a geometric series. Furthermore, the expected total discounted reward can be shown to be equivalent to an expected total undiscounted reward problem but where the horizon $T$ is random.

**c.** Instead of introducing a discount factor to keep the sum bounded, we can also take an average of the reward over time. This yields the so-called *average reward* criterion:

$$g_\pi(s) = \lim_{T\to\infty} \frac{1}{T} \mathbb{E}\left[\sum_{t=0}^{T} r(S_t, U_T) | S_0 = s\right] = \lim_{T\to\infty} \frac{1}{T} v_T^\pi(s)$$

We refer to the function $g_\pi$ as the *gain* of the policy $\pi \in \Pi^{\text{HR}}$.

Going back to the classic discounting case and letting $r_0 = 0$ and $P_\pi^0 \equiv I$, where $r_0$ and $P_\pi^0$ represent the initial reward and the transition probability matrix for all states at $t = 0$ respectively, we can express the equation 3.1.8 in vector notation as

$$v_\gamma^\pi = \sum_{t=0}^{\infty} \gamma^t P_\pi^t r_{d_t} \tag{3.1.10}$$

$$= r_{d_1} + \gamma P_{d_1} r_{d_2} + \gamma^2 P_{d_1} P_{d_2} r_{d_3} + ... \tag{3.1.11}$$

$$= r_{d_1} + \gamma P_{d_1} (r_{d_2} + \gamma P_{d_2} r_{d_3} + \gamma^2 P_{d_2} P_{d_3} r_{d_4} + ...) \tag{3.1.12}$$

$$= r_{d_1} + \gamma P_{d_1} v_\gamma^{\pi'} \tag{3.1.13}$$

where $\pi' = (d_2, d_3, ...)$. The last equation shows that the discounted reward corresponding to policy $\pi$ equals the discounted reward in a one-period problem in which the decision maker uses decision rule $d_1$ in the first period and receives the expected total discounted reward of policy $\pi'$ as a terminal reward. The component-wise notation can be expressed as

$$v_\gamma^\pi(s) = r_{d_1}(s) + \sum_{j \in S} \gamma p_{d_1}(j|s) v_\gamma^{\pi'}(j) \tag{3.1.14}$$

The two last equations are valid for any $\pi \in \Pi^{\mathrm{MR}}$; however, when $\pi$ is stationary so that $\pi' = \pi$, they simplify further. Let $d^\infty \equiv (d, d, ...)$ denote the stationary policy which uses decision rule $d \in D^{\mathrm{MR}}$ at each decision epoch. Our expected total discounted reward vectorial and component-wise equations (3.1.13 and 3.1.14) then respectively become

$$v_\gamma^{d^\infty} = r_d + \gamma P_d v_\gamma^{d^\infty} \tag{3.1.15}$$

$$v_\gamma^{d^\infty}(s) = r_d(s) + \sum_{j \in \mathcal{S}} \gamma p_d(j|s) v_\gamma^{d^\infty}(j) \tag{3.1.16}$$

The value function $v_\pi$ is then the unique solution to a linear system of equations which we will refer to as the *policy evaluation equations* [1]. From an operator-theoretic point of view, this is to say that $v_\pi$ is the fixed-point of some operator $L_\pi$ defined by the linear transformation:

$$L_\pi v := r_\pi + \gamma P_\pi v \ .$$

For $v_\pi$ to be a fixed-point of[2] $L_\pi$, this means that $v_\pi$ is an element of $\mathcal{V}$ with the property that $L_\pi v_\pi = v_\pi$.

[1] In the RL literature, these are generally called *Bellman equations*, which we prefer to use only for the equations arising from Bellman's optimality principle.

[2] We define a fixed-point with respect to its operator, hence our usage of *of*.

A fundamental result about $L_\pi$ relates the recursive form of the policy evaluation equations with that of the corresponding *Neumann series*:

$$\begin{aligned} v_\pi &= \sum_{t=0}^{\infty} (\gamma P_\pi)^t r_\pi \\ &= (I - \gamma P_\pi)^{-1} r_\pi \\ &= r_\pi + \gamma P_\pi v_\pi \ . \end{aligned}$$

This equality holds when the spectral radius of $\gamma P_\pi$ is strictly less than 1. The spectral radius of some square matrix $A \in \mathbb{R}^{m \times m}$ is defined as $\sigma(A) := \max_{1 \leq i \leq m} |\lambda_i|$ where $\lambda_i$ is an eigenvalue of $A$. This is true since $P$ is a stochastic matrix (summing across columns gives a vector of ones) and $\|\gamma P\| = \gamma \|P\| < 1$ assuming that $\gamma \in [0, 1)$.

This in turns follows from the fact that $\sigma(A) \leq \|A\|$ for any square matrix $A$. To see this, note that by definition any $\lambda_i$ satisfies $Avx_i = \lambda_i vx_i$ where $vx_i$ is the eigenvector associated with $\lambda_i$. Taking the norm on both sides, we have [3] $\|A\ vx_i\| = \|\lambda_i\ vx_i\| \leq \|A\| \|vx_i\|$. Therefore, if $\lambda^\star = \max_{1 \leq i \leq m} |\lambda_i|$, then $|\lambda^\star| \|x^\star\| \leq \|A\| \|vx^\star\|$ and $\sigma(A) \leq \|A\|$.

## 3.2. Optimal Solutions in the Infinite-Horizon Setting

The optimality equations for the total expected discounted reward are defined as:

$$v^*(s) = \max_{u \in \mathcal{U}_s} r(s, u) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u) v^\star(s') \ . \tag{3.2.1}$$

Correspondingly, the *Bellman optimality operator* is defined as:

$$Lv := \max_{d \in D^{\mathrm{MD}}} r_d + \gamma P_d v \ . \tag{3.2.2}$$

The fact that we can restrict our attention to deterministic Markovian rules follows from

$$\max_{d \in D^{\mathrm{MD}}} r_d + \gamma P_d v = \max_{d \in D^{\mathrm{MR}}} r_d + \gamma P_d v \ . \tag{3.2.3}$$

Because Markovian deterministic decision rules are a subset of Markovian randomized decision rules, we have:

$$\max_{d \in D^{\mathrm{MD}}} r_d + \gamma P_d v \leq \max_{d \in D^{\mathrm{MR}}} r_d + \gamma P_d v \ . \tag{3.2.4}$$

To obtain the other direction:

[3] Given that we have a *consistent* norm.

$$\max_{d \in D^{\mathrm{MD}}} r_d + \gamma P_d v \geq \max_{d \in D^{\mathrm{MR}}} r_d + \gamma P_d v \ , \tag{3.2.5}$$

we write:

$$\max_{u \in \mathcal{U}_s} r(s,u) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u) v(s') \geq \sum_{u \in \mathcal{U}_s} \pi(u|s) \left( r(s,u) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u) v(s') \right) \ . \tag{3.2.6}$$

In order to show that the Bellman optimality equations have a solution, we can use Banach fixed-point theorem and establish that $L$ is a *contraction mapping*. An operator $T : \mathcal{U} \to \mathcal{U}$ (from a Banach space to a Banach space) is a contraction mapping if we can show that there is a $\lambda \in [0,1)$ such that:

$$\|Tv - Tu\| \leq \lambda \|v - u\| \ , \tag{3.2.7}$$

for $u \in \mathcal{U}$ and $v \in \mathcal{U}$. The Banach fixed-point theorem then tells us two things:

(1) There exists a unique fixed-point.

(2) We can get to this fixed-point by computing a sequence $v^{(t+1)} = Tv^{(t)} = T^n v^{(0)}$ starting from some arbitrary $v^{(0)}$

To establish that $L$ is a contraction mapping, it is easier to switch to the component form of the optimality equations and write:

$$(Lv)(s) - (Lu)(s) = \left( \max_{u \in \mathcal{U}_s} r(s,u) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u) v(s') \right) - \left( \max_{u \in \mathcal{U}_s} r(s,u) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u) u(s') \right) \ . \tag{3.2.8}$$

Now let $u_s^* := \mathrm{argmax}_{u \in \mathcal{U}_s} r(s,u) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u) v(s')$ and assume that $Lv \geq Lu$. We then have:

$$0 \leq (Lv)(s) - (Lu)(s) \tag{3.2.9}$$

$$\leq r(s,u_s^*) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u_s^*) v(s') - r(s,u_s^*) - \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u_s^*) u(s') \tag{3.2.10}$$

$$= \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u_s^*) \left( v(s') - u(s') \right) \leq \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u_s^*) \|v - u\| = \gamma \|v - u\| \ . \tag{3.2.11}$$

The last step follows from the fact that we assume that we use the infinity norm (in finite dimensional spaces). The same result can also be obtained if we start with the assumption that $Lu \geq Lv$. If we arrive at the same conclusion assuming either $(Lv)(s) \geq (Lu)(s)$ or $(Lu)(s) \geq (Lv)(s)$, then we might as well take the absolute value and have:

$$|(Lv)(s) - (Lu)(s)| \leq \gamma \|v - u\| \ . \tag{3.2.12}$$

Taking the maximum on the left hand side then leads to:

$$\|Lv - Lu\| \leq \gamma \|v - u\| \ . \tag{3.2.13}$$

While Banach fixed-point theorem helps us establish the existence of a solution to the optimality equations (the optimal value function), the question still remains of determining a corresponding optimal policy. Having established that we could take the supremum over deterministic decision rules rather than the larger set of randomized decision rules, we have yet to say something about the kind of policies that can be obtained. In particular, we would like to not only have deterministic randomized decision rules, but also stationary policies. The gist of this statement revolves around the notion of *conserving* decision rules, which means that for some Markovian deterministic decision rule $d$:

$$L_d v^\star := r_d + \gamma P_d v^\star = v^\star \ . \tag{3.2.14}$$

It is important to note that this definition involves $L_d$, the policy evaluation operator, and not the optimality operator $L$. Put more simply, if a decision rule is conserving, if we were to apply the policy evaluation operator of this decision rule to the optimal value function then we would obtain back the optimal value function. Markovian deterministic decision rules which satisfy this equality are also said to be $v^\star$-improving. In general, a Markovian deterministic decision rule is $v$-improving[4] (with respect to a given $v$) if:

$$L_d v = Lv = \max_{d \in \mathcal{D}} r_d + \gamma P_d v \ , \tag{3.2.15}$$

meaning that the decision rule picks actions according to the argmax on the right-and-side.

### 3.2.1. Value Iteration

The value iteration algorithm follows directly from the constructive nature of the Banach fixed-point theorem: that of applying a contraction mapping repeatedly starting from an arbitrary guess of the fixed-point. But the Banach fixed-point theorem is also not fully computational in the sense that it cannot just be translated naively to an algorithm as it involves composing the contraction mapping with itself infinitely. Alternatively from heuristically stopping whenever the difference between two iterates is bellow a defined threshold, we can devise a reasonable stopping criterion which can guarantee that we

[4]In the reinforcement literature, we would rather talk of a *greedy* policy.

terminate close enough to the true solution.

At every iteration, the *value iteration* algorithm computes:

$$v^{(t+1)}(s) = \max_{u \in \mathcal{U}_s} r(s,u) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,u) v^{(t)}(s') \ . \tag{3.2.16}$$

It can then be shown that if we interrupt the process whenever $\|v^{(t+1)} - v^{(t)}\| < \frac{\epsilon(1-\gamma)}{2\gamma}$, then the error must be $\|v^{(t+1)}) - v^\star\| < \frac{\epsilon}{2}$. This bound follows from the fact that:

$$\|v^{(t+1)} - v^\star\| \leq \frac{\gamma}{1-\gamma} \|v^{(t+1)} - v^{(t)}\| \ , \tag{3.2.17}$$

where $\pi_\epsilon$ is the stationary deterministic policy derived from the value function upon terminating value iteration. Therefore this inequality states that upon termination (by the above criterion), the error cannot be larger than the difference between the last two iterates scaled by $\gamma(1-\gamma)^{-1}$.

Similarly, we can show that:

$$\|v_{\pi_\epsilon} - v^{(t+1)}\| \leq \frac{\gamma}{1-\gamma} \|v^{(t+1)} - v^{(t)}\| \ , \tag{3.2.18}$$

where $\pi_\epsilon$ is the stationary policy obtained upon terminating. Remember that we choose to stop when:

$$\|v^{(t+1)} - v^{(t)}\| < \frac{\epsilon(1-\gamma)}{2\gamma} \iff \frac{\gamma}{1-\gamma} \|v^{(t+1)} - v^{(t)}\| < \frac{\epsilon}{2} \ . \tag{3.2.19}$$

This means that both $\|v^{(t+1)} - v^\star\| < \epsilon/2$ and $\|v_{\pi_\epsilon} - v^{(t+1)}\| < \epsilon/2$, which can be combined together via the triangle inequality:

$$\|v_{\pi_\epsilon} - v^\star\| \leq \|v_{\pi_\epsilon} - v^{(t+1)}\| + \|v^{(t+1)} - v^\star\| \leq \epsilon \ . \tag{3.2.20}$$

Upon termination, the algorithm would have found a policy whose value function is within $\epsilon$ of the optimal value function $v^\star$.

### 3.2.2. Policy Iteration

Rather than applying the method of successive approximation directly over the space of value functions, the policy iteration algorithms refines an initial guess on an optimal policy

by evaluating its performance and adjusting it accordingly.

If $d_t$ is the $t$-th decision rule produced by policy iteration, then the *policy evaluation* step consists in solving for $v$ in $(I - \gamma P_{d_t})v = r_{d_t}$ which then guides the *policy improvement* step in which $d_{t+1}$ is generated [5] by taking $d_{t+1} \in \text{argmax}_{d \in D} r_d + \gamma P_d v^{(t)}$. Because of the partial order on $\mathcal{V}$, the policy improvement step can be computed equivalently as $d_{t+1}(s) \in \text{argmax}_{u \in \mathcal{U}_s} r(s,u) + \gamma \sum_{s'} p(s'|s,u) v^{(t)}$ and not be a direct enumeration over all $d \in D$. This process is repeated (from an arbitrary $d_0$) until two successive policies are the same.

A key property in the analysis of policy iteration is to establish that the sequence $\{v^{(t)}\}$ is monotonic, that is $v^{(t+1)} \geq v^{(t)}$. To see this, note that the next decision $d_{t+1}$ has been generated from $v^{(t)}$ by taking $d_{t+1} \in \text{argmax}_{d \in D} r_d + \gamma P_d v^{(t)}$. Therefore, this $d_{t+1}$ being the maximum over all $d \in D$ – including $d_n$ – then it holds that:

$$r_{d_{t+1}} + \gamma P_{d_{t+1}} v^{(t)} \geq r_{d_t} + \gamma P_{d_t} v^{(t)} = v^{(t)} \ . \tag{3.2.21}$$

The equality follows from the fact that we make sure to solve for $v^{(t)}$ during the policy evaluation step. We then have:

$$r_{d_{t+1}} + \gamma P_{d_{t+1}} v^{(t)} \geq v^{(t)} \iff r_{d_{t+1}} \geq \left(I - \gamma P_{d_{t+1}}\right) v^{(t)} \ . \tag{3.2.22}$$

By multiplying on both sides of the inequality, we get:

$$(I - \gamma P_{d_{t+1}})^{-1} r_{d_{t+1}} \geq v^{(t)} \ , \tag{3.2.23}$$

which means that $v^{(t+1)} \geq v^{(t)}$.

Because the sequence of values $v^{(t)}$ generated by policy iteration are non-decreasing and that the improvement step is taken over the finite set of deterministic decision rules, then policy iteration must terminate in a finite number of steps. By requiring that $d_{t+1} = d_t$ upon termination, then we have that $v_{d_{t+1}} = v_{d_t}$ and so:

$$v^{(t)} = v^{(t+1)} = r_{d_{t+1}} + \gamma P_{d_{t+1}} v^{(t+1)} = r_{d_{t+1}} + \gamma P_{d_{t+1}} v^{(t)} = \max_{d \in D} r_d + \gamma P_d v^{(t)} \ . \tag{3.2.24}$$

This means that $v^{(t)}$ satisfies the Bellman optimality equations and that we have found an optimal policy. The policy iteration algorithm therefore provides us with a constructive

[5] Also referred to as a *greedification* step in RL.

proof for the existence of a solution to the Bellman optimality equations which is not based on the contraction mapping theorem.

We use the term *generalized policy iteration* (GPI) to refer to the general idea of letting policy-evaluation and policy-improvement interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy. If both processes stabilize, then a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation holds, and thus that the policy and the value function are optimal.

## 3.3. Approximate Solution Methods

While the analytical background of all the material introduced in this chapter is rigorously accurate, most real-world applications involve additional challenges such as huge dimensional spaces and introduce complexities that the classical framework cannot harness with unless using additional resources. So far we presented our attempt of a concise and synthesized version of what we would define as the *foundations* of classical sequential decision making, but operations research, control theory and several other disciplines developed (and are still developing) an enormous quantity of very diversified algorithms and methodologies, which exploit a lot of different strategies and numerical tools that are way beyond the scope of this work.

Among popular methodologies, to name only a few, we can think of approximate dynamic programming, linear programming methods (involving, or not, numerical approximations), constrained optimization, integer programming and many more. Two of the most important tools we pay particular attention on are leveraging trajectory samples collected by experience, and using function approximators to replace the full computation and avoid storing specific quantities. Considering this, we now put a halt to the general decision making theory to turn our focus towards the realm of statistical learning and simulation. We will then see in the third section how we propose to merge specific material from both of these two fields to leverage the benefits of both worlds in an elegant and efficient manner.

# Part 2

# STATISTICAL LEARNING

# Chapter 4

# Machine Learning Fundamentals

Statistical learning refers to a vast set of tools and methods for understanding, or more specifically, *learning*, from data. Practically, and from an application perspective, it can been seen as the scientific study of algorithms and statistical models used to perform a specific task by relying on examples rather than explicit instructions. *Machine learning* (ML) can be seen as a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decrease emphasis on proving confidence intervals around these functions.

Statistical learning tasks are typically divided in three main categories, which often fuse and overlap in a variety of aspects:

(1) **Supervised learning**: statistical model creation for predicting an output based on one or more inputs.
(2) **Unsupervised learning**: considering inputs but no supervising output; learning and extracting relationships and structure from data.
(3) **Reinforcement learning**: optimal control framework, where training information is used to evaluate actions, in order to maximize a numerical signal defined in accordance to a specific goal.

To practical ends, we only briefly introduce the unsupervised learning setting and focus the vast majority of this chapter on supervised learning. The primary purpose is to present the general basic concepts of Machine Learning, which will dress the table for deep learning and reinforcement learning, the main subjects of the next chapters. The content developed herein mainly originates from the *An Introduction to Statistical Learning* [**35**], *The Elements of Statistical Learning* [**30**], *Deep Learning* [**27**] and *Stochastic Simulation* [**41**] books, in addition to available distributed academic material.

## 4.1. Learning Algorithms

As previously stated, statistical learning refers to algorithms and tools able to learn from data. But such a definition is very unformal and does not allow for a rigorous identification. To resolve this issue, we use the explicit definition as defined by [**46**]:

> «*A computer program is said to learn from experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with experience $E$.* »

One can imagine a wide variety of such $E$, $T$ and $P$, and we do not attempt to formally define the nature or extensiveness of these entities. For our present interest, we only present the two most popular and most widely used tasks, which are the *regression* and the *classification*. Machine learning tasks can be described in terms of how the learning system should process a sample of data. We refer to the inherent measured properties of the sample as a collection of quantitative *features*. We typically represent this sample as a vector $x \in \mathbb{R}^a$ where each component $x_i$ of the vector is a specific feature.

*Classification* is defined as a task where the learning algorithm is asked to specify which of $k$ categories some input belongs to. To solve this problem, the general approach is to produce a function $f : \mathbb{R}^a \to \{1,...,k\}$. When $y = f(x)$, the model assigns an input described by vector $x$ to a category identified by a numeric code $y$. Some variants of the classification task can also output a probability distribution $p_y$[1] over classes instead.

*Regression* is the task where the learning algorithm is asked to predict a numerical value given some input. Typically, the learning algorithm is asked to output a mapping $f : \mathbb{R}^a \to \mathbb{R}^b$, where $b$ is the dimension of the output. It resembles the classification task a lot, differing only in the format and continuous nature of the target.

To evaluate quantitatively the performance of machine learning algorithms on a specific task, one must use a performance measure $P$. For classification, the typical measure is the *accuracy* of the model, defined as the proportion of examples for which the model produces the correct output:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{4.1.1}$$

[1]By strict mathematical definition, a *function* can take many arguments as input but is constrained to a single output. Consequently, in this case, we would rather speak of a *mapping* from $x$ to a probability distribution $p_y$.

Equivalent information could also be obtained by measuring the *error rate*, i.e. the proportion of examples for which the model produces an incorrect output.

In the regression setting, it is typical to base the performance evaluation metric on the difference between the real sampled value, and the predicted one produced by the learning algorithm. This approach yields the popular *mean absolute error* (MAE), *mean squared error* (MSE), *root mean squared error* (RMSE), *log mean squared error* (LMSE) and *mean absolute percentage error* (MAPE), defined respectively as:

$$\frac{1}{n}\sum_{i=1}^{n}|y^i - \hat{y}^i| \quad \{\text{ MAE }\}\,, \tag{4.1.2}$$

$$\frac{1}{n}\sum_{i=1}^{n}(y^i - \hat{y}^i)^2 \quad \{\text{ MSE }\}\,, \tag{4.1.3}$$

$$\frac{1}{n}\sqrt{\sum_{i=1}^{n}(y^i - \hat{y}^i)^2} \quad \{\text{ RMSE }\}\,, \tag{4.1.4}$$

$$\frac{1}{n}\sum_{i=1}^{n}\log\left(y^i - \hat{y}^i\right)^2 \quad \{\text{ LMSE }\}\,, \tag{4.1.5}$$

$$\frac{100}{n}\sum_{i=1}^{n}\frac{y^i - \hat{y}^i}{y^i} \quad \{\text{ MAPE }\}\,. \tag{4.1.6}$$

where $i$ indexes the total number of samples, $n$. In most application scenarios, we are interested in the *generalization* performance of the learning algorithm, i.e. its performance on data unseen before. We therefore evaluate these performance measures using a subset of data referred to as the *test set*, separated from the rest of the data the algorithm is trained on.

Machine learning algorithms can be broadly categorized as *unsupervised* or *supervised* by what kind of experience they are allowed to have during the learning process. Most the algorithms implied in the context of this work, in the supervised setting, can be understood as being allowed to experience an entire *dataset*. A dataset is simply defined as a collection of available samples. Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target. The usage of the term *supervised* originates from the view of the target $y$ being provided by an instructor or teacher

who shows the machine learning system what to do, unlike the unsupervised learning setting.

Supervised and unsupervised learning are not formally defined terms. One can show using joint distribution decomposition that both are not completely formal or distinct concepts, but rather simply help roughly categorizing some methodologies with machine learning algorithms. Traditionally, people refer to regression, classification and structured output problems as supervised learning. Other variants of the learning paradigms are possible. For example, in semi-supervised learning, some examples include a supervision target but other do not. In reinforcement learning, as we will see in the last chapter of this section, the learning algorithm interacts with an environment, so there is a feedback loop between the learning system and its experiences.

## 4.2. Estimator, Bias and Variance

Given a probability space $(\Omega,\mathcal{F},\mathbb{P})$, a *random variable* $X$ is an $\mathcal{F}$-measurable[2] function $X : \Omega \to \mathbb{R}$ that assigns a real number $X(\omega)$ to each outcome $\omega \in \Omega$. We typically refer to this number as $X$ before its true value is known.

We define the *mathematical expectation* (or *theoretical average*) of a real-valued random variable $X$ by:

$$\mathbb{E}[X] = \int_{\Omega} X(\omega)\mathbb{P}(d\omega) \ . \tag{4.2.1}$$

If $X$ is discrete, and given its mass function $p$, this general expression transforms to

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} x_i \ p(x_i) \ , \tag{4.2.2}$$

while for the continuous case with density $f$ we have

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x \ f(x)dx \ . \tag{4.2.3}$$

From this, we introduce the *variance* of $X$ as :

$$\mathrm{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \ . \tag{4.2.4}$$

Very often denoted $\sigma^2(X)$, the variance is a strictly positive quantity that can be interpreted as the moment of inertia of the probability distribution function (PDF) or probability mass function (PMF) of $X$ with respect to its mean. Taking the square root of the variance yields the *standard deviation*, and the ratio :

[2] $\{\omega : X(\omega) \in B\} \in \mathcal{F} \ \forall \ B \in \mathcal{B}$, where $\mathcal{B}$ is the Borel $\sigma$-field on $\mathbb{R}$.

$$\frac{\sqrt{\text{Var}[X]}}{\mathbb{E}[X]} = \frac{\sigma(X)}{\mathbb{E}[X]} \ , \tag{4.2.5}$$

is referred to as the *coefficient of variation* or *relative error*.

The *covariance* between two random variables $X$ and $Y$ is defined by

$$\text{Cov}(X,Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] \ . \tag{4.2.6}$$

In the case where $X$ and $Y$ are independent, by *modus ponens* we have $\text{Cov}(X,Y) = 0$. It is important to note that the reverse statement is not necessarily true, i.e, a null covariance does not necessarily guarantee independence between two variables. We also introduce the (Pearson) linear correlation coefficient between $X$ and $Y$:

$$\rho(X,Y) = \text{Corr}(X,Y) = \frac{\text{Cov}(X,Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}} = \frac{\mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]}{\sigma(X)\sigma(Y)} \ . \tag{4.2.7}$$

The linear correlation coefficient is always within the bound $[-1,1]$, and can be viewed as a standardized version of the covariance, since it measures the linear dependence between $X$ and $Y$. Furthermore, we say that $X$ and $Y$ are *uncorrelated* if $\rho(X,Y) = 0$, *positively correlated* if $\rho(X,Y) > 0$, and *negatively correlated* if $\rho(X,Y) < 0$.

### 4.2.1. Point Estimation

Experience samples or *stochastic simulation* are very often used to estimate unknown mathematical expectations over a random variable $X$. *Point estimation* refers to the estimation of an unknown quantity $\nu$ by some random variable $X$ with expectation $\mu = \mathbb{E}[X]$ and variance $\sigma^2 = \text{Var}[X]$. We introduce the *bias*, defined as the difference between $\mu$ and $\nu$:

$$\text{Bias } = \mu - \nu \ , \tag{4.2.8}$$

and we consequently say that the estimator is *unbiased* if such quantity is null.

Let $X_1,...,X_n$ denote $n$ independent realizations of the random variable $X$. Each of these realizations represents a *sample* that has the independent and identically distributed (i.i.d.) property. The *sample mean* or *empirical mean* of these realizations, also referred to as the *crude Monte Carlo estimator*, is defined by

$$\hat{\mu}_n = \bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i \ , \tag{4.2.9}$$

and their *sample variance* by

$$S_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2 = \frac{1}{n-1}\sum_{i=1}^{n} X_i^2 - \frac{n(\bar{X}_n)^2}{n-1} \ , \tag{4.2.10}$$

which yields

$$\mathbb{E}[\bar{X}_n] = \mu \ , \mathbb{E}[S_n^2] = \sigma^2 \ . \tag{4.2.11}$$

Consequently, $\bar{X}_n$ and $S_n^2$ are unbiased estimators of $\mu$ and $\sigma^2$.

The variance of $\bar{X}_n$ is $\sigma^2/n$ and it can be estimated by $S_n^2/n$. If $(X_1,Y_1)$,...,$(X_n,Y_n)$ is an i.i.d sample of $(X,Y)$, then an unbiased estimator of $\mathrm{Cov}[X,Y]$ is given by

$$\widehat{\mathrm{Cov}}[X,Y] = \frac{1}{n-1}\sum_{i=1}^{n}(X_i,\bar{X}_n)(Y_i - \bar{Y}_n) \ . \tag{4.2.12}$$

An infinite sequence of estimators $\{Y_n, n \geq 1\}$ is often denoted simply by $Y_n$. For example, we sometimes use $\bar{X}_n$ and $S_n^2$ to denote infinite sequences indexed by $n$. With this abuse of notation, when $n \to \infty$, we say that $Y_n$ is *asymptotically unbiased* if $\mathbb{E}[Y_n - \mu] \to 0$, *consistent* if $Y_n \to \mu$ in probability, and *strongly consistent* if $Y_n \overset{w.p.l.}{\to} \mu$.

## 4.2.2. Confidence Intervals

It is common practice to assess the accuracy of $\bar{X}_n$ as an estimator of $\mu$ via a *confidence interval* (CI). More formally, a random interval $[I_1,I_2]$ is a CI at confidence level $1-\alpha$ (or equivalently a $100(1-\alpha)\%$ CI) for $\mu$ if $\mathbb{P}[I_1 \leq \mu \leq I_2] = 1-\alpha$. The boundaries $I_1$ and $I_2$, along with the *width* $I_2 - I_1$, are all random variables.

We define the *coverage error* as the difference

$$\mathbb{P}[I_i \leq \mu \leq I_2] - (1-\alpha), \tag{4.2.13}$$

where the left term represents the true *coverage probability*, which differs from $1-\alpha$ and is often unknown. Consequently, an ideal CI should present small values of $\mathbb{E}[I_2 - I_1]$ and $\mathrm{Var}[I_2 - I_1]$, in addition to presenting an approximately correct coverage. Finally, as CI is said to be *asymptotically valid* if its depends on the sample size $n$, and has its coverage error converge to 0 when $n \to \infty$.

Considering a fixed confidence level $1-\alpha$ and a sufficient sample size $n$, by the central limit theorem (CLT) we have that if

$$z_{1-\alpha/2} = \Phi^{-1}(1-\alpha/2) \equiv \Phi(z_{1-\alpha/2}) = 1-\alpha/2 \ , \tag{4.2.14}$$

then

$$\mathbb{P}[|\bar{X}_n - \mu| \leq z_{1-\alpha/2} S_n/\sqrt{n}] \approx 1 - \alpha \ . \tag{4.2.15}$$

In this case, a CI at approximate level $1 - \alpha$ is given by

$$[I_{n,1}, I_{n,2}] = [\bar{X}_n - z_{1-\alpha/2} S_n/\sqrt{n},\ \bar{X}_n + z_{1-\alpha/2} S_n/\sqrt{n}] \ . \tag{4.2.16}$$

For example, setting $\alpha = 0.05$ yields $z_{1-\alpha/2} \approx 1.96$.

As $n \to \infty$, the width of the CI is asymptotically proportional to $\sigma/\sqrt{n}$, resulting in a convergence in the order of $O(n^{-1/2})$.

### 4.2.3. Efficiency of Simulation Estimators

Considering an estimator $X$ available to estimate some unknown quantity $\mu$, it is often useful to have a notion of efficiency for simulation estimators that takes into account both the work and the noise. In addition to the bias, variance, MSE and RMSE introduced previously, we also define the *relative error* RE of $X$

$$\mathrm{RE}[X] = \frac{\mathrm{RMSE}(X)}{|\mu|},\ \mu \neq 0, \quad , \tag{4.2.17}$$

which is typically more relevant when $\mu$ is very small.

Introducing the expected computational resources required to compute $X$ as a random variable $C(X)$, usually correlated with $X$, we call the product $C(X)\ MSE(X)$ the *work-normalized* MSE of $X$ and we subsequently define the *efficiency* of $X$ as the inverse

$$\mathrm{Eff}(X) = \frac{1}{C(X)\ \mathrm{MSE}(X)} \ . \tag{4.2.18}$$

An estimator $Y$ is said to be more efficient than another estimator $X$ if $\mathrm{Eff}(Y) > \mathrm{Eff}(X)$. Efficiency improvement means finding a more efficient estimator $Y$ than the currently used estimator $X$ in the above sense. The ratio $\mathrm{Eff}(Y)/\mathrm{Eff}(X)$ is the efficiency improvement factor. Often, both estimators are unbiased and are assumed to have roughly the same computational costs; in such case, improving the efficiency is equivalent to reducing the variance. For this reason, one often speaks of *variance reduction techniques* (VRTs). However, efficiency can sometimes be improved by increasing the variance (slightly), while reducing the computing cost.

Lastly, if the computing cost is not taken into account, e.g. if two estimators require about the same computing effort, we call $\mathrm{Var}[X]/\mathrm{Var}[Y]$ the *variance reduction factor* (VRF) of $Y$

with respect to $X$. It represents the factor by which the variance is reduced when using $Y$ instead of $X$.

### 4.2.4. Curse of Dimensionality

Machine learning tasks become exceedingly difficult as the number of dimensions in the data grows, and as the density of samples becomes consequently lower and lower. This phenomenon is known as the *curse of dimensionality*, and induces a statistical challenge because the number of possible configurations becomes much larger than the training examples. Considering this, it is often useful or necessary to perform *feature selection* or *dimensionality reduction* to reduce the number of inputs in our learning algorithms.

## 4.3. Model Selection and Data Manipulations

One of the main challenges in statistical learning is the performance on new, previously unseen inputs. The ability to perform well on these out of training samples is called *generalization*.

Starting from a quantitative or categorical realization, we consider a target variable $Y$, an input vector $X$, and a prediction model $\hat{f}(X)$ estimated from a training set $\mathcal{T}$ that is trying to approximate the underlying predictive relationship between the inputs and the outputs. Typically, for a given learning algorithm, we compute some error measure or *loss function* $L(Y,\hat{f}(X))$ to assess the difference between the two quantities. We then proceed to reduce this quantity during the training process, using the *training error* defined as the average loss over the training sample:

$$\overline{\text{err}} = \frac{1}{N}\sum_{i=1}^{N} L(y_i.\hat{f}(x_i)) \ . \tag{4.3.1}$$

But what differentiates this classical optimization problem from machine learning is the *generalization error*, or *test error*, defined as the prediction error over an independent test sample

$$\text{Err}_{\mathcal{T}} = \mathbb{E}[L(Y,\hat{f}(X))|\mathcal{T}] \ , \tag{4.3.2}$$

where both $X$ and $Y$ are drawn randomly from their joint population. Further extending our analysis beyond a single fixed training set, we can introduce a related quantity called the *expected test error*

$$\text{Err} = \mathbb{E}[L(Y,\hat{f}(X))] = \mathbb{E}[\text{Err}_{\mathcal{T}}] \ , \tag{4.3.3}$$

where the expectation averages over everything random, including the randomness in the training set that produced $\hat{f}$.

As the model grows in complexity, it uses the training data more and is able to adapt to more complicated underlying structures, which leads to a decrease in bias but an increase in variance. Some intermediate model complexity can give a minimum expected test error. Unfortunately, training error is not a proper estimate of the test error since it consistently decreases with model complexity, typically reaching 0 with high enough capacity. However, a model with zero training error is overfit to the training data and will typically generalize poorly. Consequently, we can conclude that the two main factors determining how well a machine learning algorithm will perform are its ability to:

(1) Make the training error small.
(2) Make the gap between training and test error small.

If we consider that the available data arose from a statistical model of the form :

$$Y = f(X) + \epsilon \ , \tag{4.3.4}$$

where $\mathrm{Var}(\epsilon) = \sigma_\epsilon^2$, and where the random error $\epsilon$ has $\mathbb{E}(\epsilon) = 0$ and is independent of $X$. Furthermore, $f(x) = \mathbb{E}[Y|X = x]$, and we assume that the conditional distribution $\Pr(Y|X)$ depends on $X$ only through the condition mean $f(x)$. We can derive the following expression for the expected prediction error of a regression fit $\hat{f}(X)$ at an input point $X = x_0$ using the squared-error loss:

$$\begin{aligned}
\mathrm{Err}(x_0) &= \mathbb{E}[(Y - \hat{f}(x_0))^2 | X = x_0] \\
&= \sigma_\epsilon^2 + [\mathbb{E}(\hat{f}(x_0)) - f(x_0)]^2 + \mathbb{E}[\hat{f}(x_0) - \mathbb{E}(\hat{f}(x_0))]^2 \\
&= \sigma_\epsilon^2 + \mathrm{Bias}^2(\hat{f}(x_0)) + \mathrm{Var}(\hat{f}(x_0)) \\
&= \text{Irreducible Error } + \mathrm{Bias}^2 + \text{ Variance } .
\end{aligned}$$

The first term is the variance of the target around its true mean $f(x_0)$, and cannot be avoided no matter how good the estimate of $f(x_0)$ is, unless $\sigma_\epsilon^2 = 0$. The second and third term represent respectively the bias and the variance, as introduced previously. The last equalities illustrate a widely known and fundamental phenomenon in statistical learning typically referred to as the *Bias-Variance trade-off*. This trade-off shows the conflict in trying to simultaneously minimize the bias and the variance, which are the two sources of error preventing supervised learning algorithms to generalize beyond their training set.

### 4.3.1. Capacity, Overfitting and Underfitting

Reducing the training error, and minimizing the gap between training and test error corresponds to the two central challenges in machine learning: *underfitting* and *overfitting*. The former arises when the learning algorithm is not able to obtain a sufficiently low error value on the training set, while the latter arises when the gap between the training error and test error is too large.

The *capacity* of a model, i.e. its ability to fit a wide variety of functions, is an aspect that we can control and which will affect overfitting or underfitting tendency behaviours. Models with low capacity may struggle to fit the training set, while model with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set. One way to control the capacity of a learning algorithm is by choosing its hypothesis space, that is, the set of functions that the learning algorithm is allowed to select as being the solution. Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with.

Quantifying the capacity of the model enables statistical learning theory to make quantitative predictions. One of the most important results is that it can be shown that the discrepancy between training error and generalization error is bounded from above by a quantity that grows as the model capacity grows but shrinks as the number of training examples increases. While these bounds provide intellectual justification that machine algorithms can work, they are rarely used in practice, especially in deep learning where quantifying the capacity of a neural network is especially difficult because of its optimization algorithm.

While simpler functions are more likely to generalize, the chosen hypothesis must be sufficiently complex to achieve low training error. This reality leads to the typical phenomena where training error decreases until it asymptotes to the minimum possible error value, as the model capacity increases.

Let us imagine an omniscient model that simply knows the true probability distribution that generates the data. Even in this eventuality, some error will incur from the fact that there may still be some noise in the distribution, either because the mapping from $\boldsymbol{x}$ to $y$ in the supervised learning case is inherently stochastic, or because $y$ may be a deterministic function that involves other variables besides those included in $\boldsymbol{x}$. We refer to the resulting

error in the omniscient case as the *Bayes error*.

Training and generalization error vary as the size of the training set varies. Expected generalization error can never increase as the number of training examples increases. For nonparametric models, more data yields better generalization until the best possible error is achieved. On the parametric setting, any fixed model with less than optimal capacity will asymptote to an error value that exceeds the Bayes error.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting. When generalization error is measured by the MSE, increasing capacity tends to increase variance and decrease bias. This is qualitatively illustrated in figure 4.1.

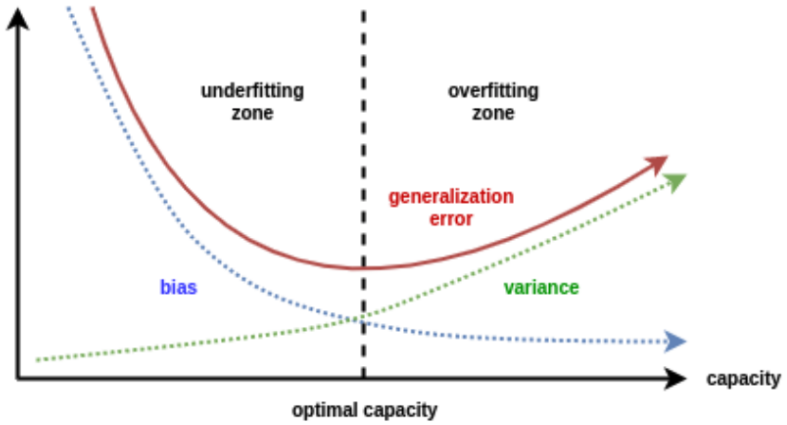


**Fig. 4.1.** Illustrative representation of the bias-variance tradeoff as a function of model capacity [**60**].

As a general rule, as we use more flexible methods, the variance will increase, and the bias will decrease. As we increase the flexibility of a class of methods, the bias tends to initially decrease faster than the variance increases, reducing the test MSE. However, at some point increasing flexibility has little effect on the bias but starts to significantly increase the variance. When this happens, the test MSE increases.

### 4.3.2. Hyper-Parameters and Datasets

Most machine learning algorithms simply experience a dataset. A dataset can be described in many ways. In all cases, a dataset is a collection of examples which are in turn collections of features. One common way to represent a dataset is with a *design matrix*. A design matrix contains a different sample in each row, and each column of the matrix corresponds to a different feature. This allows us to represent the whole dataset with a

design matrix $X \in \mathbb{R}^{n \times m}$, where $n$ is the number of samples and $m$ the number of features. It is important to note that if there is a time relation along the rows of our matrix, i.e. each row represents a progressive observation of our features at a different time step, we refer to our dataset as a *time series*. Lastly, if our samples are not all of the same size, e.g. sentences or pictures, then we refer to our dataset as a set containing $n$ elements.

We introduce the notion of *hyperparameters*, that is, algorithm's settings we can use to control their behavior. Unlike the parameters, the value of hyperparameters are not adapted by the learning itself (though a nested learning procedure could be created). Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because of the difficulty of the optimization. More frequently, the setting must be a hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies to all hyperparameters that control model capacity: if learned on the training set, such entities would always choose the maximum possible model capacity, resulting in overfitting.

It is important to recall the two possible separate goals we might have in mind as machine learning practitioners: a) *model selection*, i.e. estimating the performance of different models in order to choose the best one (b) *model assessment*, i.e. having chosen a final model, estimating its prediction error (generalization error) on new data. If enough data samples are available, the best approach for both problems is to randomly divide the dataset (considering examples do not have any dependency between them, like in a time series) into three parts: a *training set*, a *validation set* and a *test set*. The first one is used to fit the models, while the second and last one are respectively used to estimate the prediction error for model selection and to assess the generalization error of the final chosen model. It is important that the test examples are not used in any way to make choices about the model, including its hyperparameters. There is no general rule on the proportions to choose for the three sub-datasets, as it depends on the signal-to-noise ratio of the underlying function, as well as the complexity of the models being fit, but 50%,25% and 25% are typical proportions used respectively for training, validating and testing.

Dividing our dataset into three subcomponents can be problematic, as it reduces the amount of available data for training, validating and testing. Furthermore, if the validation and test sets are too small, statistical uncertainty arises around the estimated average test error. To this end, for datasets that are not very large, an alternative procedure can be made at the price of increased computational cost. The idea of this procedure is to repeat the training and testing computation on different randomly chosen subsets or splits of the original dataset. The most common of these is the $k$-fold cross-validation (CV) procedure, in which a partition of the dataset is formed by splitting it into $k$ nonoverlapping subsets. Let

$\kappa : \{1,...N\} \to \{1,...K\}$ be an indexing function indicating one of the $K$ partitions to which observation $i$ among $N$ observations is allocated by the randomization. Denote by $\hat{f}^{-k}(x)$ the fitted function computed with the $k$th part of the data removed. Then, the cross-validation estimate of prediction error is:

$$\mathrm{CV}(\hat{f},\alpha) = \frac{1}{N}\sum_{i=1}^{N} L(y_i,\hat{f}^{-\kappa(i)}(x_i)). \tag{4.3.5}$$

The specific case $K = N$ is referred to as *leave-one-out* CV.

### 4.3.3. Parametric vs Non-Parametric

Function estimators, depending on their nature, can be further divided into two classes: parametric and non-parametric.

*Parametric* models learn a function described by a parameter vector whose size is finite and fixed before any data is observed. Parametric methods involve a two-step model-based approach:

(1) Make an assumption about the functional form or shape of the function (for example, linear in $X$).
(2) Define procedure to fit or train the model.

Parametric methods assume underlying statistical distributions in the data. Therefore, several conditions of validity must be met so that the result of a parametric test is reliable. They present the advantage of fixed complexity, that remains constant even as the number of samples grows, but the disadvantage that the chosen model will usually not match the true unknown form of the function we wish to approximate.

*Non-parametric* methods seek an estimate of the function that gets as close to the data points as possible without being too rough or wiggly. Nonparametric methods are more robust than parametric tests - i.e. they are valid in a broader range of situations (fewer conditions of probability). They present the advantage of avoiding any assumption of a particular form for the function, with the potential to fit accurately a wide range of different shapes. On the other end, such methods present a growing number of effective parameters, require a larger number of observations than parameters to obtain more accurate estimates, and have increasing complexity as a function of the training set size. Lastly, parametric approaches will usually outperform the non-parametric ones if the parametric form that has been selected is close to the true form of our function.

### 4.3.4. Regularization

Our modern ideas about improving the generalization of machine learning models relies on a principle typically referred to as *Occam's razor*. This principle states that among competing hypotheses that explain known observations equally well, we should choose the "simplest" one. This idea was formalized and made more precise in the twentieth century by the founders of statistical learning theory [**27**].

The *no free lunch theorem* for machine learning states that, averaged over all possible data-generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. In other words, there is no machine learning algorithm universally better than any other. Fortunately, if we make assumptions about the kinds of probability distributions we encounter in real-world applications, then we can design learning algorithms that perform well on these distributions. Thus, the goal of machine learning research is not to seek a universal learning algorithm, but instead to understand what kind of distributions are relevant to the real world that an agent experiences, and what kind of machine learning algorithm perform well on data drawn from the kinds of data-generating distributions we care about.

The behavior of our algorithms is strongly affected not just by how large we make the set of functions allowed in its hypothesis space, but also by the specific identity of those functions. For example, linear regression has a hypothesis space consisting of the set of linear functions of its input. But if we consider problems that behave in a very nonlinear fashion, then this class of algorithms will yield very poor results. We can thus control the performance of machine learning algorithms by choosing what kind of functions we allow them to draw solutions from, and their amount.

Another idea is to give a learning algorithm a preference for one solution over another in its hypothesis space. Such approach is used in *weight decay*, where we minimize a sum comprising both our training error and a criterion that expresses a preference for the weights to have a smaller norm. We shall cover this in details in the next chapter.

Formally, we define *regularization* as any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization.

## 4.4. Maximum Likelihood Estimation

To formalize the aforementioned discussions, and to efficiently derive specific functions that are good estimators for different models, we introduce in this section one of the most popular frameworks in machine learning: the *maximum likelihood* principle.

Let us consider a set of $n$ i.i.d sample realizations from a random variable $X$, denoted $\mathcal{X} = \{X_1, X_2, ..., X_n\}$. Let $p_{\text{model}}(X;\theta)$ be a parametric family of probability distributions over the same space indexed by $\theta$, i.e. a mapping any configuration $X$ to a real number estimating the true probability $p_{\text{data}}(X)$. The maximum likelihood estimator for $\theta$, denoted $\theta_{ML}$ is then defined as

$$\theta_{\text{ML}} = \arg\max_{\theta} p_{\text{model}}(\mathcal{X};\theta), \tag{4.4.1}$$

$$= \arg\max_{\theta} \prod_{i=1}^{n} p_{\text{model}}(X_i;\theta) \tag{4.4.2}$$

To avoid the inconvenience of the product over many probabilities, we prefer to take the logarithm of the likelihood, which does nots change its arg max but does conveniently transform the product into a sum:

$$\theta_{\text{ML}} = \arg\max_{\theta} \sum_{i=1}^{n} \log p_{\text{model}}(X_i;\theta) \tag{4.4.3}$$

Lastly, as we stated before, since the arg max does not change we can simply divide by $n$ to obtain a version of the criterion that is expressed as an expectation with respect to the empirical distribution $\hat{p}_{\text{data}}$ defined by the training data:

$$\theta_{\text{ML}} = \arg\max_{\theta} \mathbb{E}_{X\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(X_i;\theta) \tag{4.4.4}$$

Maximum likelihood thus seeks to find the optimum values for the parameters by maximizing a likelihood function derived from the training data. Another way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution, defined by the training set, and the model distribution where the degree of dissimilarity between both is measured by the Kullback-Leibler (KL) divergence. The KL divergence is given by

$$D_{\text{KL}}(\hat{p}||p_{\text{model}}) = \mathbb{E}_{X\sim\hat{p}_{\text{data}}}[\log \hat{p}_{\text{data}}(X) - \log p_{\text{model}}(X)]. \tag{4.4.5}$$

Since the left term is a function only of the data-generating process, it means that upon training the model to minimize the KL divergence, we need only minimize the expectation

on the second term, which is the same as equation 4.4.4. Another interpretation is that we can see our problem as an attempt to make the model distribution match the empirical distribution, since our ideal true data-generating distribution is not available. Despite the same optimal $\theta$ arising whether we are maximizing the likelihood or the KL divergence, it is important to note that the values of the objective functions are however different.

The maximum likelihood estimator can readily be generalized to estimate a conditional probability $P(y|x;\theta)$ in the supervised learning setting, in order to predict $Y$ given $X$. Denote by $\mathcal{X}$ all our inputs, and $\mathcal{Y}$ all our observed targets, then the conditional maximum likelihood estimator is

$$\theta_{\text{ML}} = \arg\max_{\theta} P(\mathcal{Y}|\mathcal{X};\theta). \tag{4.4.6}$$

Furthermore, with the i.i.d. assumptions we can rewrite

$$\theta_{\text{ML}} = \arg\max_{\theta} \sum_{i=1}^{n} \log P(Y_i|X_i;\theta). \tag{4.4.7}$$

We finish this section with the main properties of the maximum likelihood. One of its main appeals is that it can be shown that as the number of examples $n \to \infty$, it is the best estimator asymptotically in terms of its rate of convergence. Furthermore, under appropriate conditions, the maximum likelihood estimator has the property of consistency, i.e. as the number of training examples approaches infinity, the maximum likelihood estimate of a parameter almost surely converges to the true value of the parameter.

## 4.5. Bayesian Statistics

The *frequentist* and *Bayesian* methods are the two most common approaches in statistics. While until now we focused on the frequentist setting, Bayesian statistics instead express a *degree of belief*, which may be based on prior knowledge inherent from results or personal beliefs. Up until now we considered estimating a single value of $\theta$, which we considered to have a fixed but unknown value, then making all predictions thereafter based on that one estimate which was considered a random variable being a function of the dataset. In the Bayesian setting, we can instead consider all possible values of $\theta$ when making a prediction, that is, we now consider the true parameter $\theta$ to be unknown or uncertain and thus represented as a random variable.

Before observing the data, we use the *prior probability distribution*, or simply *prior* $p(\theta)$ to represent our knowledge of $\theta$. Typically, we start from a high entropy broad distribution,

and we further seek to converge and concentrate around a few highly likely values of the parameters. Now using our set of data samples $\{x^{(1)},...,x^{(n)}\}$, we can recover the effect of data on our belief about $\theta$ by combining the data likelihood $p(x^{(1)},...,x^{(n)}|\theta)$ with the prior via Bayes' rule:

$$p(\theta|x^{(1)},...,x^{(m)}) = \frac{p(x^{(1)},...,x^{(m)}|\theta)p(\theta)}{p(x^{(1)},...,x^{(m)})} \tag{4.5.1}$$

We previously described how the frequentist approach addresses the uncertainty in a given point estimate by evaluating its variance. The Bayesian equivalent is to simply integrate over it, which tends to protect well against overfitting. Lastly, Bayesian methods typically generalize much better when limited training data is available but typically suffer from high computational cost when the number of training examples is large, which makes it a good candidate for grid search methodologies where the amount of samples is relatively low due to long calculation training.

# Chapter 5

# Deep Learning

Deep learning is an important sub-field of machine learning encompassing any calculation implying artificial neural networks (ANN). Neural networks are computing systems vaguely inspired by the biological brain structure, structured as alternating layers of aggregated parameters and nonlinear activation functions. The power of neural networks comes from their ability to learn complex highly non-linear representations of different abstraction levels from data samples, and to scale particularly well with massive datasets. In their modern form, ANNs have existed since the 1980s when LeCun et al. successfully trained a convolutional neural network to recognize handwritten zip codes [**40**]. Since 2012, deep learning has been successfully applied to many different problems and has contributed to state-of-the-art results in a wide range of fields including computer vision, machine translation, natural language processing and speech synthesis. To this day, deep learning is probably one of the most powerful function approximation technique readily available for a broad range of applications.

In the present chapter, we introduce all the major components involved in deep learning and try to provide the reader with a mainly qualitative and sometimes partially quantitative understanding of every concept at stake. Our goal is to introduce all the important elements used and manipulated in the third part of this thesis. A reader interested in the full rigorous, detailed development and implementation of the topics presented herein should refer to the specific resources provided locally for the sake of completeness. The vast majority of the content in this chapter comes from the classic ressource *Deep Learning* [**27**] book, as well as publications related to specific topics.

## 5.1. Deep Feedforward Networks and Generalities

Deep feedforward neural networks, also called multi-layer perceptrons (MLPs) represent the archetype and foundation block of deep learning. Their name originates from the simple

unilateral flow of information during the mapping process of a variable $y$ from an input sample $x$. When *feedback* connections are included in the structure of the neural network, i.e. outputs of the model are fed back into itself, we refer to such entity as *recurrent neural networks* instead.

### 5.1.1. Structure and Forward Pass

Despite their biological analogy, ANNs are called *networks* because of the chain of functions they constitute. More formally, an MLP can be seen as $f(x) = f^{(n)}(f^{(n-1)}(...f^{(2)}(f^{(1)}(x))))$, where we define $f^{(1)}$ as the *first layer* or *input layer*, $f^{(i)}$ as the $i-1$th *hidden layer*, and $f^{(n)}$ as the *output layer*. Furthermore, we say that the length of the chain defines what we call the *depth* of the model. It is this terminology that inspired the name *deep learning*. Furthermore, layers are typically vector valued. Consequently, we refer to the dimensionality of these as the *width* of the structure. As we will see, both of these lastly introduced parameters play a major role in the capacity of our model.

Considering the previous development, a neural network layer can either be seen as a single vector-to-vector mapping, or as many parallel units computing a vector-to-scalar function, where those elements can be interpreted as a loose analogy of a neuron. More formally, an ANN architecture provides a parametric class of functions where each layer applies a linear transformation of the form :

$$A(x) + b \ , \tag{5.1.1}$$

on a given vector input $x$, where $A$ is an $m \times n$ matrix representing the *weights* of this specific layer and $b \in \mathbb{R}^m$ is the bias vector. $A$ and $b$ are *parameters* of the network, i.e. quantities to be determined by the training process.

Each of the $m$ scalar output components of the linear layer then becomes the input to a nonlinear differentiable and monotonically increasing *activation function* $\sigma$. The simplest and most popular activation in modern practices is the *rectified linear unit* (ReLU) defined as :

$$\text{ReLU} \equiv \max\{0,x\} \ , \tag{5.1.2}$$

Even if by strict mathematical definition the function is not differentiable at 0, since the loss rarely reaches 0 because most of the time training converges to a local minima, and since at the implementation one can simply choose either the left or right side derivative to apply at 0, ReLU is one of the most efficient[1] and widely used functions in practice. Close

[1]Its computation requires only a multiplication, and its derivative is one of the simplest expressions.

variants like the *leaky ReLU* or *exponential linear unit* (ELU) are also popular, since they respectively allow gradient flow for negative values and analytical differentiability. Other functions, used since the early days of neural networks, have the property :

$$-\infty < \lim_{x\to-\infty} \sigma(x) < \lim_{x\to\infty} \sigma(x) < \infty \ . \tag{5.1.3}$$

Such functions are called *sigmoids*, and some common choices are the *hyperbolic tangent* function

$$\sigma(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \ , \tag{5.1.4}$$

and the logistic function

$$\sigma(x) = \frac{1}{1 + e^{-x}} \ . \tag{5.1.5}$$

Several examples of these functions are shown in figure 5.1.

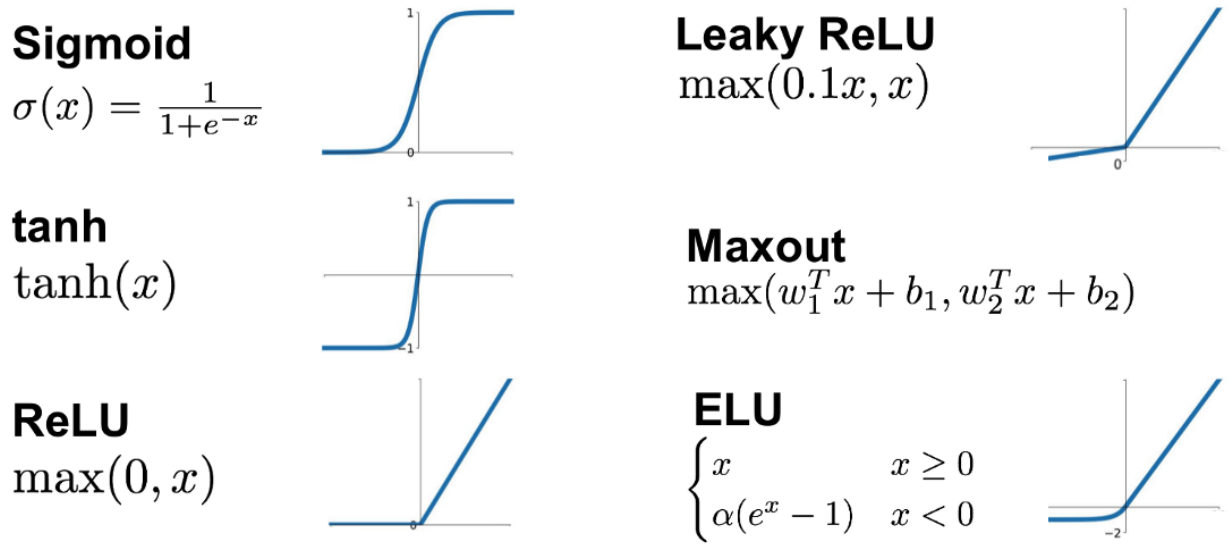


**Fig. 5.1.** Illustration of six of the most popular activation functions used in deep learning [**34**].

Many other activation functions are frequently used, and even today that choice remains an active area of ongoing research. However, for our present purpose, we do not need to dive in further details. In what follows, we will ignore the character of the function $\sigma$ (except for differentiability), and simply refer to it as a "nonlinear unit" and to the corresponding layer as a "nonlinear layer."

The outputs of the linearities can be seen as features of the original input, or as a new representation for it. Any feature that can be of practical interest can be produced or be closely approximated by a neural network. The only requirement is a feedforward network

with a linear output layer and at least one hidden layer with non-linear activation functions, which can approximate any Borel measurable[2] function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. This is a consequence of the *universal approximation theorem* (UAT) [**68**]. A neural network may also approximate any function mapping from any finite dimensional discrete space to another[3].

The concatenation of successive linear and non-linear functions, particularly when the ouputs of each nonlinear layer becomes the inputs of the next linear layer, create what is known today as a *deep* neural network architecture. More formally, we define the *forward pass* of an ANN made of $S$ successive non-linear layers as the whole computation from the inputs $x$ to the output $\hat{y}$ expressed as:

$$\{ \text{ Input Layer } \} \quad h^{(1)} = \sigma^{(1)}\Big(W^{(1)\mathrm{T}}x + b^{(1)}\Big) \ , \tag{5.1.6}$$

$$\{ \text{ Hidden Layers } \} \quad h^{(i)} = \sigma^{(i)}\Big(W^{(i)\mathrm{T}}h^{(i-1)} + b^{(i)}\Big) \ \forall \ i \in [2,...,S] \ , \tag{5.1.7}$$

$$\{ \text{ Output Layer } \} \quad \hat{y} = \sigma_o^{(S+1)}\Big(W^{(S+1)\mathrm{T}}h^{(S)} + b^{(S+1)}\Big) \ . \tag{5.1.8}$$

where we purposely added the subscript $o$ to the last activation function to emphasize that this layer usually has its own specific activation different from the others, depending on the nature of the machine learning task at stake. Practically speaking, it is very common to use a simple linear activation on the output layer for regression, and *softmax* activation for classification, since it can be interpreted as a probability distribution considering its mathematical properties:

$$\text{softmax} = \frac{\mathrm{e}^{x_j}}{\sum_k \mathrm{e}^{x_k}} \tag{5.1.9}$$

where the $j$ and $k$ indexes act on the $x$ vector's individual components.

Considering the UAT, one can reasonably question the utility of having multiple layers. Two main reason justify this point:

(1) Considering the outputs of each non-linear layer as features, multilayer networks thus provide a sequence of features, where each set of features in the sequence is a function of the preceding one of the sequence. This produces a hierarchy of features, which

[2]The development of Borel measurability is beyond the scope of this work; to all practical purposes, we instead consider the simplified statement "any continuous function on a closed and bounded subset of $\mathbb{R}^n$."
[3]It is important to note here that despite the ability of a neural network to *represent* any function given enough hidden units, we have no guarantees that it will actually *learn* that function.

for specific applications, can be exploited to specialize the role of some of the layers and to enhance particular characteristics of the input. Furthermore, deeper networks were also shown to have the potential of being exponentially more efficient [**49**].

(2) Given the presence of multiple linear layers, one may consider the possibility of using matrices $A$ with a particular sparsity pattern or other structure that embodies special linear operations such as convolution (see next section). When such structures are used, the training problem often becomes easier, because the number of parameters in the linear layers is drastically decreased.

It is worth noting that while in the early days of neural networks practitioners tended to uses few non-linear layers, more recently a lot of success in certain problem domains (including image and speech processing, as well as approximate DP) has been achieved with deeper architectures.

### 5.1.2. Back-Propagation and Training

As we just detailed the information flow through a neural network from the input to the output, called the *forward propagation*, we now introduce the *back-propagation* algorithm [**58**] (often simply called *backprop*). In this setting, we consider the result of the forward pass from which we compute a scalar cost $\mathcal{L}(\theta)$, and consider how we can now inversely allow the information from the cost to flow backward through the network for training. More formally, recalling the maximum likelihood principle, the training problem of a neural network has the form:

$$\min_{\theta} \sum_{k=1}^{n} \Big( \mathcal{L}\big(\hat{y}^{(k)}(x^{(k)};\theta), y^{(k)}\big) + \lambda \Omega(\theta) \Big) \ , \tag{5.1.10}$$

where $\theta$ represents the collection of all the parameters in the neural network, $\mathcal{L}$ a cost function (usually MSE for regression), $\hat{y}$ and $y$ the estimated and real value of some desired quantity, $\Omega$ an additional regularization penalty (see section 4.3.4 on regularization), and $\lambda$ a size parameter for such penalty. This problem is an unconstrained nonconvex differentiable optimization problem that can in principle be addressed with any of the standard gradient-type methods.

Let us consider a sum of component functions:

$$f(\theta) = \sum_{i=1}^{m} f_i(\theta) \ , \tag{5.1.11}$$

with $f_i$ a differentiable scalar function of the $n$-dimensional parameter vector $\theta$. The classical gradient method would generate a sequence of $\{\theta^k\}$ iterates, starting from an initial starting point $\theta_0$ usually chosen to be an initial guess to the minimum of $f$:

$$\theta^{k+1} = \theta^k - \gamma^k \nabla f(\theta^k) = \theta^k - \gamma^k \sum_{i=1}^{m} \nabla f_i(\theta^k) \ , \tag{5.1.12}$$

where $\gamma^k$ represents the *step size* parameter. Similarly, the *incremental gradient method* only uses the gradient of a single component of $f$ at each iteration:

$$\theta^{k+1} = \theta^k - \gamma^k \nabla f_{i_k}(\theta^k) \ , \tag{5.1.13}$$

for which $i$ indexes the set $\{1,...,m\}$, and is chosen by some deterministic or randomized rule. The selection method for such indexes is important for the performance of the method, and yields different possible algorithms for example whether a cyclic order, a uniform random order or a cyclic order with random reshuffling is chosen. Finally, it is important also to compare the performance of the incremental and nonincremental settings depending on their distance from convergence: when far from convergence, incremental methods tend to be much faster, particularly if $m$ is large; when close to convergence, incremental methods can be inferior, in particular since the ordinary gradient method can be shown to converge with a constant stepsize under reasonable assumptions [**10**].

We now introduce the *stochastic gradient descent* (SGD) algorithm, a type of incremental gradient methods, to minimize a function $f(\theta) = \mathbb{E}\{F(\theta,w)\}$, with $F(\bullet,w) : \mathbb{R}^n \to \mathbb{R}$:

$$\theta_{k+1} = \theta^k - \gamma^k \nabla_\theta F(\theta^k,w^k) \ , \tag{5.1.14}$$

where $w^k$ is a sample of $w$, a random variable and $\nabla_\theta F$ represent the gradient of $F$ with respect to $\theta$. This method has a rich theory and a long history, and is strongly related to the classical algorithmic field of *stochastic approximation* [**12, 57**]. While we can see direct link with incremental gradient methods by viewing the expected value of $F$ as a weighted sum of cost function components, SGD is inherently different in the sense that it involves stochastic sampling instead.

Reverting back to ANN training, among all gradient-type methods, *incremental aggregated gradient methods* are still the most used in practice at the time of writing:

$$\theta^{k+1} = \theta^k - \gamma^k \sum_{l=0}^{m-1} \nabla f_{i_{k-l}}(\theta^{k-l}) \tag{5.1.15}$$

where $f_k$ is the new component function selected for iteration $k$. The addition to the incremental gradient method is that here we use the sum of the component gradients

computed in the past $m$ iterations, which is an approximation to the total cost gradient $\nabla f(\theta^k)$. The general idea around this method is to reduce the error between the estimation of the true gradient and the incrementally computed approximation which yields faster computation. Under certain conditions, this method exhibits a linear convergence rate, without incurring the cost of a full gradient evaluation at each iteration (a strongly convex cost function and with a sufficiently small constant stepsize are required for this [**11**]).

Let $L_1$,...,$L_{S+1}$ represent the matrices representing the *linear* layers, so that the output of $L_1$ is given by the vector $L_1 x$ and at the $k$th layer ($k > 1$) we obtain $L_k\ h_{k-1}$, where $h_{k-1}$ is the output of the precedent *non-linear* layer $k-1$. We can then express the output of an MLP as

$$\hat{y}(L_1,...,L_{S+1},x) = L_{S+1} h_S\left(L_S ... h_1(L_1 x)...\right) \quad . \tag{5.1.16}$$

Using the previous development, we can now synthesize the generic steps to fully train an ANN, regardless of the chosen gradient method algorithm:

(1) Apply forward propagation to calculate sequentially the outputs of the linear layers that will be needed in the previous equation, and to derive the error vector .
(2) Use a backward pass through the network to calculate sequentially the partial derivative of the cost function with respect to every component, using the chain rule. This step is the back-propagation.
(3) Apply the chosen gradient-type method, SGD for example.

## 5.2. Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a specialized kind of ANN, explicitly designed for data with specific topologies. For example, time series data present a 1-D relationship along the time dimension, images a 2-D spatial relationship, videos a 3-D spatial and temporal relationship etc. CNNs are one of the most successful tools in practical machine learning applications to this day. To put it simply, this type of ANN uses a convolution operation in place of general matrix multiplication in at least one of their layers, which allows them to efficiently deal with high-dimensional inputs, to exploit different types of topologies, and to have built-in invariance to certain variations we could expect from a same quantity under different observations.

### 5.2.1. The Convolution Operation

Very popular and wide-spread in the digital and signal processing communities, the convolution is an operation on two functions of a real-valued argument. More formally speaking,

the convolution of such two functions $f$ and $g$, denoted using the convolution operation $*$, is defined as the integral of the product of the two functions after one is reversed and shifted:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \ , \tag{5.2.1}$$

which, by engineering convention, is equal to $f(t) * g(t)$.

In the usual machine learning terminology, the first function $f$ is referred to as the *input*, and the second function $g$ as the *kernel*. The output is sometimes, and usually in machine learning, referred to as the *feature map*. Considering the discrete setting, like in many application scenarios where one only has access to data samples, we can define the discrete convolution:

$$(f * g)(t) = \sum_{\tau=-\infty}^{\tau=\infty} f(\tau)g(t-\tau) \ . \tag{5.2.2}$$

Considering the finite aspect of the multidimensional arrays inputed in machine learning applications, the kernel being an array of parameters that are adapted by the learning algorithm and the necessity of the parameters to be explicitly stored separately, we typically assume zero values everywhere but in the finite set of available points, and can thus reduce the summation to one over a finite number of array elements.

Convolution presents the commutativity property, result of the kernel flipping relative to the input. Despite its usefulness in mathematical manipulations, such property is usually not important in neural network implementation, and a related function called *cross-correlation* is usually applied instead. It is essentially the same as the convolution operation, but without flipping the kernel. In the context of training, the learning algorithm will learn the appropriate values of the kernel inplace, whether is it flipped or not. Discrete convolution can be viewed as multiplication by a matrix, but with equality constraints on different entries. For example, in a univariate discrete convolution, a Toeplitz matrix [**84**] is typically used.

In addition to accommodating different input sizes, the convolution is used in deep neural networks because it leverages three important ideas that can help improve a machine learning system.

(1) *Sparse interactions* : a kernel smaller than the inputs requires fewer computing operations and less parameters than a traditional fully connected MLP, because of the local connectivity. Furthermore, it is possible to obtain good performance on the

machine learning task while keeping the number of units several orders of magnitude smaller than the size of the original input.

(2) *Parameter sharing* : units organized into the same *feature map* all share the same parameters, and thus, we say they have *tied weights* since the value of the weight applied to one input is tied to the value of a weight applied elsewhere.

(3) *Equivariant representation* : As a result of parameter sharing, the convolution also yields the property of equivariance to several variations we could expect, such as translation.

### 5.2.2. Pooling

A CNN layer typically consists of three consecutive elements, which are sometimes successively repeated to create deeper architectures:

(1) Several parallel convolutions, called *feature maps* produce a set of linear pre-activations where the size of the kernel is called the *receptive field*.

(2) Each pre-activation is inputed as argument into a non-linearity function such as the ReLU (see section 5.1.1). This step is sometimes called the *detector stage*.

(3) Lastly, a *pooling* function is used to further modify the output of the layer, usually to down-sample it.

Pooling layers are intended to consolidate the features learned and expressed in the preceding feature map (with non-linearity). More formally, a pooling function replaces the output of the net at a certain location with a summary statistic of the nearby inputs. For example, the *max pooling* [**85**] operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include averages, the $L^2$ norm or a weighted average based on the distance from the central pixel.

While we are constrained to stop our review on CNNs here, it is important to acknowledge the historical importance of this type of network, as they are present in almost every high-performing application scenario nowadays, and represent the daily bread and butter of a vast majority of machine learning practitioners.

## 5.3. Recurrent Neural Networks

Just like convolutional neural networks were particularly well suited to data with topological properties, recurrent neural networks [**59**] (RNNs) are a family of neural networks designed for processing sequential data of the form $x^{(1)},...,x^{(\tau)}$. We can find examples of such data in everyday life like natural language, writing, time series and many

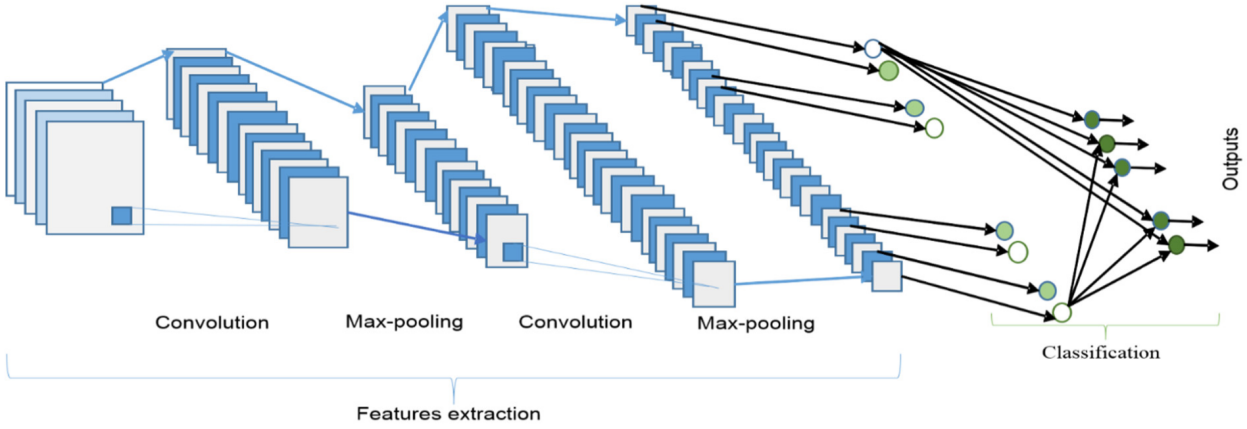


**Fig. 5.2.** The overall architecture of a Convolutional Neural Network includes an input layer, multiple alternating convolution and max-pooling layers, fully-connected layers and an output layer (classification in this case) [**4**].

others. Also in a similar fashion to CNNs, RNNs leverage the concept of parameter sharing to allow a more efficient generalization. Closely related ideas have also been developped, like using unidimensional convolutions across a temporal sequence to create what we call a *time-delay* neural networks [**39, 78**]. RNNs share parameters in a different way: each member of the output is a function of the previous members of the output. This recurrent formulation results in a parameter sharing strategy through a very deep computational graph.

In what follows, to concision ends, we use the notation $x^{(t)}$ to refer to a vector who's time step index, $t$, ranges from 1 to $\tau$. It is important to note that the $t$ index does not refer specifically to the passage of time in the real world, but simply indexes position in the given vector sequence. To fully understand RNNs, we now extend the unidirectional computational graph concept we saw with the MLP to include cycles.

### 5.3.1. Computational Graphs

Starting from the functional form of a dynamical system (see section 1.3) with state $s^{(t)}$ parametrized by $\theta$ at index $t$:

$$s^{(t)} = f(s^{(t-1)}; \theta) \ , \tag{5.3.1}$$

we define such equation as a *recurrence* since its definition at instant $t$ refers to its own definition at $t-1$. We say that we *unfold* such recurrence for a sequence of length $\tau$ if we apply repetitively the definition $\tau - 1$ times. For example, unfolding the previous equation with $\tau = 3$ yields:

$$s^{(3)} = f(s^{(2)}; \theta) \tag{5.3.2}$$

$$= f(f(s^{(1)}; \theta); \theta). \tag{5.3.3}$$

Such an expression can now be represented by a traditional directed acyclic graph if desired. Considering an additional external signal $x^{(t)}$ driving our dynamical system, the state now includes information about the whole past sequence. This idea is the fundamental principle of recurrent neural networks, which actually build an *internal state* $h$ to define the values of the hidden units. Replacing $s$ by $h$ in equation 5.3.1 and adding an external signal $x^{(t)}$ leads to:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta) \ . \tag{5.3.4}$$

Just like almost any function could be considered a feedforward neural network, essentially any function involving a recurrence can be considered a recurrent neural network.

When applied on a time series or a sequence, the recurrent neural network learns to use his hidden state $h^{(t)}$ as a partial summary of the task-relevant aspects of the past sequence of inputs up to $t$. Here we say partial because information is definitely lost since we consider a mapping from a sequence of vectors to a single fixed length vector. It is however a part of the interest, since the network *learns* the important aspects of the sequence that may be important for the nature of the task it is trained for. The most demanding situation is when we ask for the hidden state to be rich enough to allow one to approximately recover the input sequence, which yields a special type of structure called an *autoencoder*.

### 5.3.2. Recurrent Network Types

Among all the possible design patterns one can think of for neural networks, three of the most important settings are:

(1) recurrent networks producing an output at each time step, with recurrent connections between hidden units;
(2) recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step;
(3) recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output.

These patterns are presented in detail in figure 5.3.

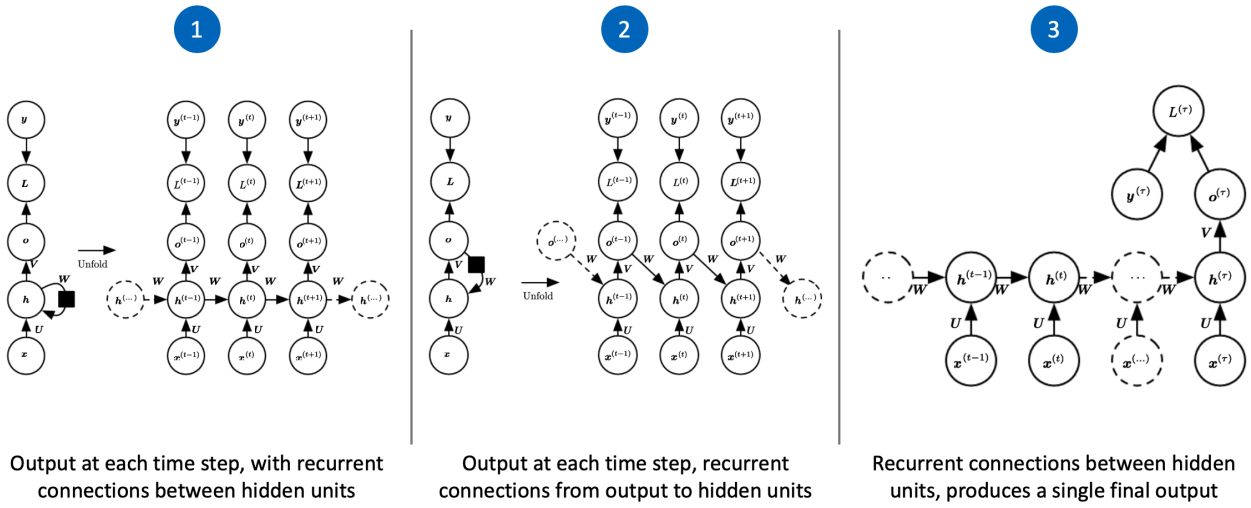


**Fig. 5.3.** Three popular recurrent neural network design patterns, with their compact computation graph (left) and its unrolled counterpart (right) [**27**].

Traditional structures considered so far only had "causal" structure, in the sense that the state at time $t$ captured only information from the past and present inputs. In many applications such as natural language processing (NLP), it is sometimes practical to consider something that depends on the whole input sequence. *Bidirectional* RNNs (or BRNNs) [**63**] were specifically invented to address that need. More specifically, BRNNs combine both an RNN moving forward through the sequence, with another RNN moving backward through it. This is pretty useful in time series to capture forward and backward dependencies in quantities variations.

Using the aforementioned material, one can easily produce an RNN mapping an input sequence to a fixed size vector, an input sequence to an output sequence of the same length, and a fixed-sized vector to a sequence. We now conclude this subsection by introducing a powerful architecture called the *encoder-decoder* [**14**] or *sequence-to-sequence* [**71**] (Seq2Seq) that allows the system to map a variable-length sequence to another variable-length sequence which can be of different size. More concretly, the framework is divided in two parts: (1) An *encoder*, *reader* or *input* RNN processes the input sequence, then emits what we call the *context* C. (2) This fixed-length vector is then used by the *decoder*, *writer* or *output* RNN to generate the desired output sequence.

### 5.3.3. Back-Propagation Through Time

Applying the training algorithm we developed earlier for MLPs to RNNs is pretty straightforward, but because the forward propagation graph is inherently sequential, the back-propagation applied to the unrolled graph cost is called *back-propagation through*

*time* (BPTT). The gradients obtained by the procedure can then be used with any general-purpose gradient-based techniques, just like in the MLP case.

Since each of the neural network's parameters must receive an update proportional to the partial derivative of the error function with respect to itself, backprop information needs to flow properly through the whole network. In many-layered feedforward networks, and in networks with longer computation graphs in general such as RNNs, the gradient flow often leads either to the *vanishing gradient* or *exploding gradient* problem. The former arises when the recurrent multiplications on numbers smaller than 1 done during the process tend to 0, while the latter inversely happens when numbers greater than 1 induce exponentially increasing numbers. While some exotic techniques have been explored, such as unsupervised pre-training of the layers [**8**], nowadays the problem of exploding gradients can be solved by *gradient clipping*, i.e. by capping the norm the gradient can take (usually to 1 in practice). On the other side, a special class of RNNs called *gated* RNNs mitigate the problem of vanishing gradients.

### 5.3.4. Long Short-Term Memory and Gated Networks

The challenge of learning long-term dependencies comes from the fact that the gradients propagated over many stages tend to either vanish, or to explode, considering the multiple multiplications of many Jacobians. To counter this problem, one of the most popular and practically applied sequence models to this day are the *gated* RNNs, which include the *long short-term memory* (LSTM) and networks based on the *Gated Recurrent Unit* (GRU). Gated RNNs incorporate a powerful internal mechanism called *gates*, usually relying on a *sigmoid* activation function because of its co-domain properties, that can regulate the flow of information. The intuition is that such gates can learn which data in a sequence is important to keep or throw away, and consequently pass relevant information down the long chain of sequences.

The LSTM presents a similar flow as a traditional RNN, but the difference lies in what we call the *cell state* and its various gates. More specifically, that cell state can be seen as an information conveyor down the sequence chain. Information then gets added or removed along the sequence, at each time step, with the following operations (see figure 5.4) within the cell itself:

(1) A **forget gate** decides which information from the previous cell state $c^{(t-1)}$ will be kept by taking the concatenation of the current input $x^{(t)}$ and the previous hidden state $h^{(t-1)}$, and inputing it through a first sigmoid function, which outputs values between 0 and 1. These outputs will then multiply the cell state in a point-wise

manner. Sigmoid output values close to 0 will "forget" and remove information, while output values near 1 will "keep" their previous cell state counterpart.

(2) An **input gate** updates the cell state by running two operations in parallel with the vector concatenation of $h^{(t-1)}$ and $x^{(t)}$: (1) the vector is inputed into a second sigmoid function to decide which values will be updated in a similar fashion to the forget gate. (2) a copy of the vector is passed through the non-linear activation function (usually a *tanh* or a $ReLU$) before being multiplied point-wise just like in the forget gate.

(3) The output of the two previous gates is then added to form the next cell state $c^{(t)}$.

(4) Lastly, an **output gate** controls the nature of the next hidden state $h^{(t)}$ by passing $h^{(t-1)} + x^{(t)}$ through a third sigmoid, which then multiplies point-wise the new cell state passed through a non-linearity to create $h^{(t)}$.

(5) Both $c^{(t)}$ and $h^{(t)}$ are then outputed and transfered to the next time step. The process is repeated for every time step.

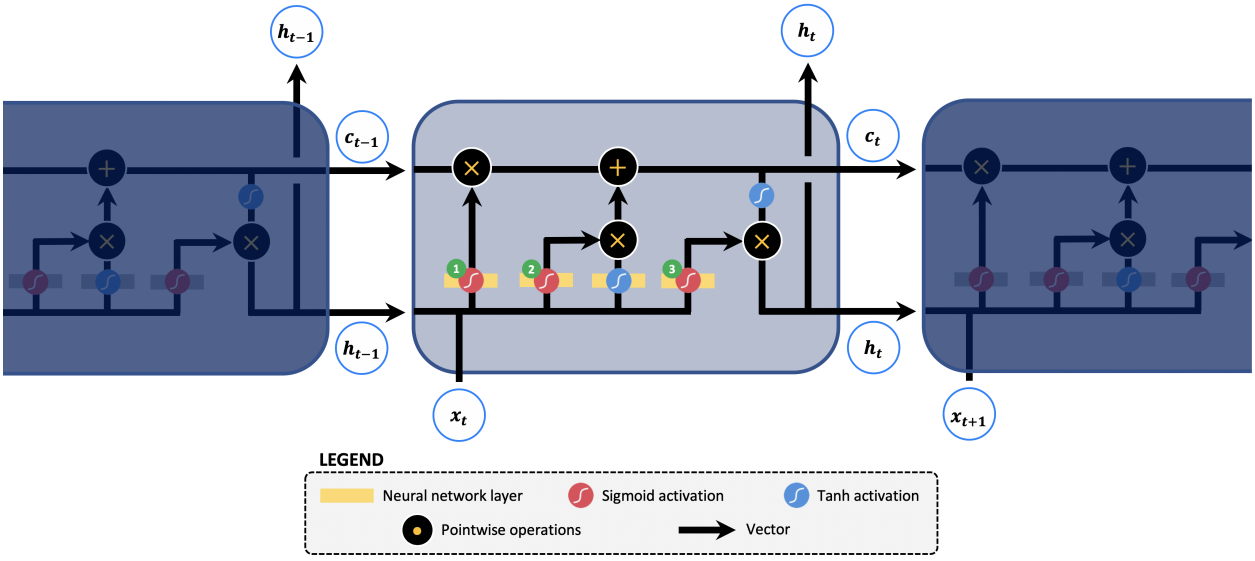


**Fig. 5.4.** Structure and basic layout of LSTM recurrent neural networks [**17**]. The cell state $c$ is represented by the top horizontal arrow, the hidden state $h$ by the bottom horizontal arrows, while the *forget*, *input* and *output* gates are denoted by the numbers 1,2 and 3 respectively.

GRUs [**15**] were introduced a lot more recently than LSTM, in 2014, and are very similar in their concept. GRUs do not rely on a cell state anymore, but still execute the same logical operations than LSTM with less gates, since now a single *update* gate simultaneously controls both the forgetting factor and the decision to update the state unit, while the *reset* gate decides how much information from the past to forget. This new type of recurrent units also has fewer tensor operations, making it a little speedier in practice.

## 5.4. Regularization

The beauty of neural networks relies in their effective combination of variegated elements. From these possible combinations arise various and continuously-evolving methodologies that allow performance improvements over the classic fundamental principles introduced earlier, and which are required to reach the state-of-the-art results we observe nowadays.

Deep neural networks are very well known for their performance scaling proportionally as they are fed larger and larger datasets, unlike many other machine learning algorithms who eventually reach a plateau. The best way to make a machine learning model generalize better is undoubtly to train on more data. A general performance improvement idea arising from this principle is to *augment* or increase (artificially or naturally) the volume of available samples in the treated dataset. We refer to such methods as *data augmentation* techniques, but we do not expand further as they are very application-specific and depend on the nature and available resources of the considered problem. We focus instead on two more general improvement possibilities : improving generalization error in the network architecture and training process, and improving the optimization algorithm itself, which tackles the challenging task of naviguating a highly non-convex optimization surface.

As introduced in chapter 4, regularization can be defined as any modifications in a learning algorithm aiming for a reduction in generalization error. In the deep learning context, regularization strategies generally imply regularizing estimators directly. In fact, unlike in many classical ML application scenarios, the model family provided in deep learning almost *never* matches the one of the real generating distribution, due to the complexity of the instances considered. To quote directly [**27**] : "*To some extent, we are always trying to fit a square peg (the data-generating process) into a round hole (our model family).*" This means that controlling the complexity of the model does not simply imply finding the right size and the right number of parameters. From an experimental perspective, it was instead found that the best fitting model is a large instance that has been regularized appropriately. We now cover a few of the most popular regularization strategies to create such large, deep regularized models.

### 5.4.1. Parameter Norm Penalties

In machine learning, many regularization approaches offer a flexible and general solution based on limiting the capacity of models by adding a parameter norm penalty $\omega(\theta)$ to the objective function $J$:

$$\tilde{J}(\theta; X, y) = J + \alpha\Omega(\theta) \tag{5.4.1}$$

where $\tilde{J}$ represents the regularized objective function, and $\alpha \in [0,\infty)$ is a *regularization rate* hyperparameter weighting the contribution of the regularization term in the objective function. For neural networks, it is usually chosen to apply a parameter norm penalty only on the weights of the affine transformation at each layer, and leave the biases unregularized since they typically require less data than the weights to fit accurately.

The *weight decay*, *ridge regression*, *Tikhonov regularization* or $L^2$ parameter norm penalty involves adding a $\Omega(\theta) = \frac{1}{2}||w||_2^2$ term to the objective function. On a per-sample basis, such modification multiplicatively shrinks the weight vector by a constant factor at each step, just before doing the gradient update. From a more general optimization perspective, introducing $H$ as the Hessian matrix of $J$ with respect to $w$ evaluated at $w^*$, the components of $w^*$ that is aligned with the $i$-th eigenvector of $H$ is rescaled by a factor of $\frac{\lambda_i}{\lambda_i+\alpha}$ where $\lambda$ is the $i$-th eigenvalue of $H$. Lastly, $L^2$ tends to push large weights down, but leaves smaller weights between 0 and 1 relatively unchanged.

In a similar fashion to weight decay, $L^1$ regularization introduces an additional $\Omega(\theta) = ||w||_1$ term to the objective function instead. However, $L^1$ regularization differs from $L^2$, since its gradient contribution no longer scales linearly with each $w_i$ but instead puts a constant factor of the same sign as $w_i$. More specifically, $L^1$ regularization results in a solution that is more *sparse*, i.e. some parameters have an optimal value of zero. Such behavior can be seen as *feature selection*, since the process chooses which subset of the available features should be used.

Lastly, it is also possible to combine different parameter norm penalties term together. One of the most common is the $L_1L_2$, combining both the $L^1$ and $L^2$ regularizations detailed above in the same instance.

### 5.4.2. Noise and Dropout

"*Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias.*" [**27**]. Noise can be applied to the inputs as a dataset augmentation strategy, but it can also be applied to the outputs or even directly to the parameters themselves. Adding noise is a practical, stochastic way to reflect uncertainty. Noise applied to the weights can also be interpreted as equivalent (under some assumptions), to encouraging stability of the function to be learned.

*Dropout* [**67**] is a modern technique consisting of randomly dropping units (along with their connections) in the neural networks during training. This prevents units from co-adapting too much, and can be interpreted as a very computation efficient way of sampling from an exponential number of different "thinned" networks. Such approach is very common in ML, usually called *bagging*. It is however important to notice the difference here, that in the dropout scenario, the models are not totally independent and trained on their own respective training set. Furthermore, several modern methods re-use this principle even in deployment to assess the variance and the uncertainty of the predictions.

## 5.5. Optimization

Neural network training is one of the most challenging optimization tasks, considering the size and complexity of the problem. It is quite common to deploy several days, or even several months of distributed computing resources to obtain interesting results. For optimization problems of this importance, specialized sets of optimization techniques have been developed, and in this section we introduce the most popular ones specific to ANN training.

### 5.5.1. Early Stopping

Training duration can be quite challenging for neural networks: too little training will yield an underfit model, while too much training could result in an overfitted model with poor performance on the test set. A compromise is to train on the training dataset, but to trigger training interruption at the point when performance on a validation dataset starts to degrade. This simple, effective, and widely used approach to training neural networks is called *early stopping*. In practice, the implementation is done by applying a prediction on the validation set after every epoch, and it is common to define a *patience* before putting a halt to the training, since validation loss can sometimes increase slightly before going down again.

### 5.5.2. Optimization Surface

We previously introduced incremental gradient methods, and more specifically the SGD algorithm. To this day, stochastic gradient descent and its variants remain among the most used optimization algorithms for machine learning, and for deep learning in particular. Such algorithms experience the challenge of navigating a strongly irregular error surface comprised of local minima, saddle points and sharp cliffs. These characteristics can respectively lead to (strongly) sub-optimal solutions, non-existing gradients or exploding gradients.

The *momentum*, from the physical analogy of oriented motion, is an algorithm accumulating an exponentially decaying moving average of past gradients, and continuing to move

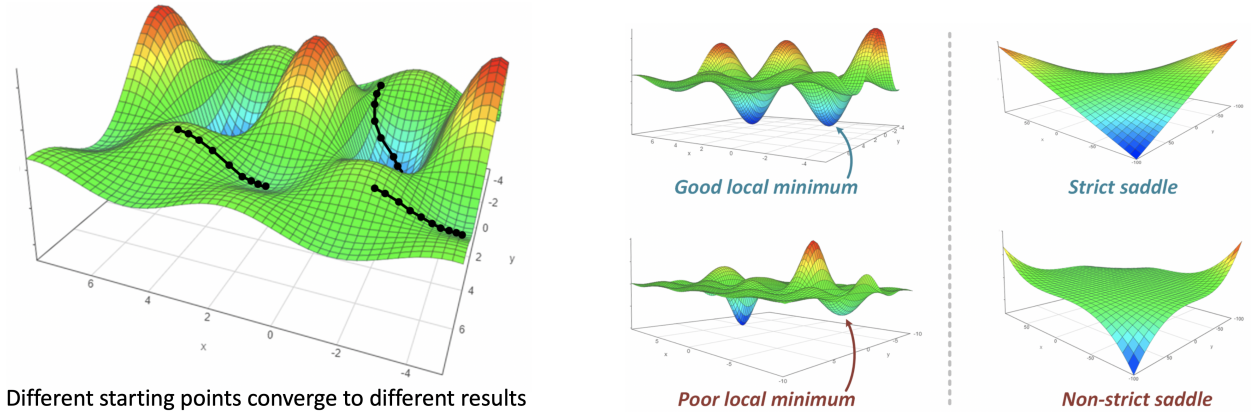


**Fig. 5.5.** Qualitative illustration of the non-convex error surface the gradient descent algorithms are applied on, as well as the inherent challenges induced by local minima and saddle points [**16**].

in their direction. Momentum is useful to solve two main problems: (1) a poor conditioning of the Hessian matrix, for example in a quadratic narrow "valley" with steep sides; (2) variance in the stochastic gradient. While the step size was previously given by the norm of the gradient multiplied by the learning rate, momentum adds a components scaling directly to how "aligned" a sequence of gradients are. For example, if the momentum algorithm always observes gradient $\mathbf{g}$, then it will accelerate in the direction of $-\mathbf{g}$ until reaching a terminal velocity.

Several variants of the momentum algorithm exist. One of the most popular is the *Nesterov momentum* [**69**], which differs from standard momentum by where the gradient is evaluated after the current velocity is applied. It can be interpreted as an attempt to add a *correction factor* to the standard method. While in the convex batch gradient case, it can be shown that the rate of convergence of the excess error is greatly improved, unfortunately in the stochastic gradient case it does not improve the rate of convergence. Nevertheless, Nesterov momentum is still often used in practice to train neural networks.

As shown in figure 5.5, the initial point for parameters drastically affects the optimization result. Considering this, the initialization strategy of the parameters in a neural network is particularly important. Surprisingly, to this day, modern initialization strategies are still simple heuristics and usually rely on achieving some useful properties since rigorous mathematical guarantees are very hard to obtain. The only important condition that has been proven a requirement is to break the "symmetry" between units. Despite many available methodologies, the one we present is one of the most commonly used in practice: the *Glorot and Bengio* [**26**] where each weight is sampled from a normalized distribution:

$$W_{i,j}\ U\Big(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\Big) \tag{5.5.1}$$

where $m$ and $n$ represent the number of units in the previous layer and the number of units in the next layer, respectively. Lastly, other simpler heuristics exist for bias initialization, but for gated recurrent neural networks it is suggested [**36**] to start close to 1.

### 5.5.3. Optimization Algorithms

The *AdaGrad* algorithm [**21**] scales the learning rate of all model parameters proportionally to the inverse of the square root of the sum of all the historical squared values of the gradient. This results in the fact that parameters with larger partial derivatives of the loss have a rapid decrease in their learning rate, while parameters with smaller partial derivatives have a relatively small decrease in their learning rate [**27**]. While AdaGrad presents some fruitful theoretical properties, in practice it tends to perform well for some models, but not for all of them.

*RMSProp* [**33**], for "root mean squared propagation" is based on AdaGrad, but modifies the gradient accumulation into an exponentially weighted moving average to perform better in the nonconvex setting. RMSProp offers a more reliable convergence once the optimization process has reached a "convex bowl", by "forgetting" the history from the extreme past that would prevent convergence. From a practital perspective, it has been empirically proven very useful and is employed routinely by deep learning practicioners.

*Adam* [**38**], derived from "adaptive moments", can be seen as a kind of combination between RMSProp and momentum, but with a few important distinctions. First, momentum is incorporated directly as an estimate of the first-order moment, with exponential weighting of the gradient. Second, Adam offers a bias correction to the estimates of both the first-order moments and the (uncentered) second-order moments to account for their initialization at the origin [**27**]. At the time of the writing, and because of his robustness to a variety of hyperparameters, Adam is the default optimizer used by the most popular deep learning libraries. Consequently, it is probably the most widely used in practice.

Lastly, despite the development and popularity of SGD and some of its variants, a lot of modern research is focused on adapting second-order methods to the neural network training problem. In contrast to first-order methods, these methods make use of second derivatives to improve optimization. However, in their crudest form such algorithms usually require the calculation and storage of the Hessian matrix, which is very cumbersome and a computational burden for most deep learning applications with millions of parameters and

thousands to millions of data samples. A computationally efficient second-order method could however revolutionize the neural network training reality.

### 5.5.4. Batch Size and Learning Rate

Two of the most crucial parameters in the aforementioned optimization techniques of the SGD family are the *batch size* and the *learning rate*. These two aspects play a key role in the gradient estimate calculated by the random sampling of such methods.

The true gradient of the total cost function gradually becomes smaller and smaller, reaching 0 when we approach a minimum. On the other hand, SGD gradient estimator introduces a source of noise with its random sampling process, that does not vanish even near a minimum. To solve this problem, the learning rate $\epsilon_k$ at iteration $k$ needs to gradually decrease over time, respecting the two following conditions to guarantee convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \ , \text{ and} \tag{5.5.2}$$

$$\sum_{k=1}^{\infty} \epsilon_k^2 < \infty \ . \tag{5.5.3}$$

In practice, it is common to adopt a *scheduled learning rate*, usually following a *learning rate decay* until iteration $\tau$:

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \ , \ \alpha = \frac{k}{\tau} \ . \tag{5.5.4}$$

Another very powerful approach is to use an *adaptive learning rate*, where in a similar fashion to early stopping (and in combination with !), we monitor validation loss plateaux, and with a patience parameter we reduce the size of the learning rate by a fixed factor (usually 2 to 10) until we reach a minimum value.

Optimization algorithms for ML typically compute each update to the parameters on an expected value of the cost function using only a subset of the terms of the full cost function. Computing the expectation is very expensive because it requires evaluating the model on every example in the entire dataset. In practice, it is common to randomly sample only a few examples from the data, i.e. a *batch* of examples, then taking the average over only those examples. Algorithms using the entire training set are called *batch* or *deterministic* gradient methods, algorithms using a single example are called *stochastic* (like SGD) or *online* methods, and other algorithms in between (most modern ML algorithms) using more than one but fewer than all the training examples are called *minibatch* or *minibatch*

*stochastic* methods[4]. The following factors influence the batch size selection:

(1) Larger batches yield a more accurate estimate of the gradient, but with less than linear returns.
(2) Hardware-wise, multicore architectures accelerate the process very little with batch sizes that are too small. On the other hand, running parallel gradients makes memory requirements scale with batch size, which is often a limiting factor for bigger datasets and full gradient estimates.
(3) Small batches were shown to offer a regularizing effect [**82**], because of the noise they add in the learning process. The high variance induced often requires a smaller learning rate to maintain stability, which results in a longer training time at the end because more steps are required to observe the entire training set.

[4]It is now common to call them simply *stochastic* methods.

# Chapter 6

# Reinforcement Learning

Reinforcement learning (RL) is the third machine learning paradigm, where training information is used to evaluate actions, rather than instruct, in order to maximize a numerical signal defined in accordance to a specific goal [**72**]. Since 2015, RL has demonstrated very impressive application results in several fields like robotics or game theory, to name only these, as it reached (super) human-level controls without any prior knowledge. This success mainly originates from the combination of reinforcement learning with deep learning as function approximator, leading to what we refer today as *deep reinforcement learning*.

As we already introduced the general framework of sequential decision-making, we directly build on the concepts presented in part I to offer a global qualitative overview of the new emerging field that is deep reinforcement learning, along with its most popular algorithms available to this day. The focus of this chapter is also to provide a detailed mathematical development for the algorithm used in the context of this work: the *Deep Q-Network* (DQN). Finally, we also try to provide a comparison with literature content from the control and operations research community to establish a link between some concepts that are mathematically equal but differ in their nomenclature.

The sources used for this chapter are tightly correlated to the ones of chapters 1 to 3. That is, Pierre-Luc Bacon's *Excursions in Reinforcement Learning* course, the *Dynamic Programming and Optimal Control (Vol I)* [**11**], *Dynamic Programming and Optimal Control (Vol II) - Approximate Dynamic Programming* [**9**], *Reinforcement Learning and Optimal Control* [**12**] books from Dimitri P. Bertsekas, the *Markov Decision Processes* [**56**] book by Martin L. Puterman, the *Reinforcement Learning* [**72**] book by Richard S. Sutton and Andrew G. Barto, and finally the *Foundations of Deep Reinforcement Learning* [**28**] book by Laura Graesser and Wah Loon Keng. We also occasionally refer to several literature articles related to specific presented content.

## 6.1. Generalities

RL relies on the framework of MDPs where an agent $a$ undergoes continuous or episodic interactions with its environment. At each sequence of discrete time steps $t$, up to a horizon $T \in [0,\infty)$, the decision maker receives a partial observation $o_t \in \mathcal{O}$ of the real state $s_t \in \mathcal{S}$ the environment $E$ is currently in, where $\mathcal{O}$ and $\mathcal{S}$ represent the discrete finite sets of observations and states, respectively. Based on the perceived observation, the controller then applies its policy $\pi$ to choose a control $u_t$ from a set of controls $\mathcal{U}$, which triggers a transition of the environment according to the probability function $P_t(s_{t+1}|s_t,u_t)$ into a new state $s_{t+1}$ and returns a new (partial) observation $o_{t+1}$ along with a reward $r_t \in \mathbb{R}$. The objective of the agent is to maximize the expected cumulative sum of these rewards on the long run. It is important to note that it is typical in RL to consider the same available action set for every epoch.

Considering the general sequential decision making theory, three primary functions can be *learned* in reinforcement learning: the value functions such as the *state-action* value function $Q^\pi(s,u)$, the *state value* function $V^\pi(s)$, or the *advantage* value function $A^\pi(s,u)$; the policy $\pi(s)$; and the environment's transition probabilities $P_t(s_{t+1}|s_t,u_t)$. Correspondingly, three major families of deep reinforcement learning algorithms arise from this idea:

(1) Value-based methods, learning value functions.
(2) Policy-based methods, learning policies.
(3) Model-based methods, learning or using a model of the environment.

Furthermore, additional families can be introduced by applying exotic combinations of these basic ideas, and are usually behind the state-of-the art results we can see at the time of the writing. In what follows, we introduce each of these primary families qualitatively and give a brief overview of one of their archetype algorithms and of its logic, except for the DQN where we explicit the full mathematical development.

A final important distinction between deep reinforcement learning algorithms, that affects how training iterations make use of the data, is whether they are *on-policy* or *off-policy*. The former implies that the algorithm can only learn from the training data generated by the current policy, which means that the data must be discarded once used for training. In contrast, *off-policy* algorithms do not abide by this requirement, and any data collected from interactions with the environment can be (re)used in training. Consequently, such methods are said to be more sample-efficient, but may require more memory to store the data.

## 6.2. Value-Based Methods and Deep Q-Network

Value-based methods rely on learning either or both of the two value function $V^\pi(s)$ and $Q^\pi(s,u)$, then using their estimate on $(s,u)$ pairs to derive a policy. Algorithms in this setting tend to be more sample-efficient than policy-based algorithms, because they have lower variance and make better use of data gathered from the environment. However, there is no guarantee that such algorithms will converge to an optimum, even local. In their standard formulation, they are usually natively limited to discrete action spaces, but can be adapted to continuous action spaces through different workarounds.

### 6.2.1. Deep Q-Network

The Deep Q-Network [**47**] is probably one of the most iconic modern value-based method. Inspired by the classical Q-Learning [**80**] algorithm, such methods are based on the state-action value function $Q$, also called *Q-factors* in the control community:

$$Q^\pi(s,u) := \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t(s_t,u_t) \,\middle|\, s_0 = s, u_0 = u\right] \quad , \tag{6.2.1}$$

and more specifically on the optimal action-value function (or optimal Q-factors) $Q^\star(s,u)$:

$$Q^\star(s,u) = \max_\pi \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t(s_t,u_t) \,\middle|\, s_0 = s, u_0 = u, \pi\right] \quad . \tag{6.2.2}$$

This last function obeys an important recursive relation known as the *Bellman equation* (see sections 2.3 and 3.2), which is based on the following intuition: if the optimal value $Q^\star(s',u')$ of the state $s_{t+1} = s'$ at the next time-step was known for all possible actions $u_{t+1} = u'$, then the optimal strategy is to select the action $u'$ maximizing the expected value of $r+\gamma Q^\star(s',u')$:

$$Q^\star(s,u) = \mathbb{E}_{s'}\left[r + \gamma \max_{u'} Q^\star(s',u') | s,u\right] \quad . \tag{6.2.3}$$

While several methodologies use this important relation, like backward induction (section 2.4), value iteration with *successive approximation* (section 3.2.1), or (generalized) policy iteration (section 3.2.2), in a similar fashion to value iteration the basic idea behind Q-learning related algorithms is to estimate the optimal action-value function by using the Bellman equation as an iterative update of the form:

$$Q_{i+1}(s,u) = \mathbb{E}_{s'}[r + \gamma \max_{u'} Q_i(s',u') | s,u] \quad , \tag{6.2.4}$$

where the expectation is typically replaced by (crude) Monte Carlo estimates (see section 4.2.1) leading to a stochastic approximation.

Such value iteration algorithms converge to the optimal action-value function, $Q_i \rightarrow Q^\star$ as $i \rightarrow \infty$. From an application perspective, this basic approach is however impractical because the action-value function is estimated separately for each trajectory, and only for the states encountered without any generalization. Instead, it is common to use a function approximator to estimate $Q(s,u,\theta) \approx Q^\star(s,u)$, leading to a family of algorithms defined as *fitted value iteration*. Using a deep neural network for doing so leads us to the basis of our DQN algorithm.

The Q-network function approximator is trained by adjusting its parameters $\theta_i$ at iteration $i$ using the mean-squared error loss from the difference obtained in equation 6.2.4, where the target values $r+\gamma \max_{u'} Q(s',u')$ are substituted with the network's approximation $y = r + \gamma \max_{u'} Q(s',u';\theta_i^-)$, using parameters $\theta_i^-$ from some previous iteration for the sake of stability[1]. This leads to a sequence of loss functions $\mathcal{L}_i(\theta_i)$ :

$$\begin{aligned} \mathcal{L}_i(\theta_i) &= \mathbb{E}_{s,u,r}\Big[(\mathbb{E}'_s[y|s,u]] - Q(s,u;\theta_i))^2\Big] \\ &= \mathbb{E}_{s,u,r}\Big[(y - Q(s,u;\theta_i))^2 + \mathbb{E}_{s,u,r}[\mathrm{Var}_{s'}(y)]\Big] \ , \end{aligned} \tag{6.2.5}$$

where $\mathbb{E}_{s,u,r}[\mathrm{Var}_{s'}(y)]$ represents the expected variance of the targets. Differentiating the loss function with respect to the weights[2] yields :

$$\nabla_{\theta_i}\mathcal{L}(\theta_i) = \mathbb{E}_{s,u,r,s'}\Big[\Big(r + \gamma \max_{u'} Q(s',u';\theta_i^-) - Q(s,u;\theta_i)\Big)\nabla_{\theta_i} Q(s,u;\theta_i)\Big] \ . \tag{6.2.6}$$

In a similar fashion to the original Q-Learning algorithm, and for computation-efficiency justification, (crude) monte carlo estimates are typically used by optimizing the loss function with incremental gradient descent algorithms such as SGD. We conclude the analogy with Q-learning by noticing that setting the update frequency of $\theta_i^-$ to be made after every time step leads us back to the original Q-Learning algorithm but using a deep neural network function approximator.

Despite this evident similarity, in addition to keeping an older copy of the weights for updates, the original DQN paper also introduced an additional concept to further improve performance and stability using neural network function approximators. *Experience replay* consists in storing the agent's experiences $e_t = (s_t,u_t,r_t,s_{t+1})$ at each time-step, in a data set or *memory buffer* $D_t = e_1,...,e_t$. During the inner loop of the algorithm, (mini-batch)

[1]This is done because in the present setting the targets depend on the network weights, unlike traditional supervised learning targets which are fixed before learning begins.
[2]The last term does not depend on the parameters $\theta_i$, and is therefore dropped.

Q-learning updates are performed, but using randomly picked (with uniform distribution) samples of experience ($s$,$u$,$r$,$s'$). This leads to the advantage that each experience sample is potentially used in many weight updates, which allows for greater data efficiency, breaks the correlations between samples and therefore reduces the variance of the updates by smoothing the distribution. Adding a gradient clipping to limit the norm of the gradient becomes also easier and further increases the learning stability. Lastly, when learning on-policy, the current parameters determine the next data sample that the parameters are trained on, which can be problematic because it introduces unwanted feedback loops. Using experience replays allow the algorithm to be *model free* and to learn *off-policy* while following another behavior policy that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an $\epsilon$-greedy policy that follows the greedy policy with probability 1-$\epsilon$ and selects a random action with probability $\epsilon$. The $\epsilon$ parameter is typically reduced progressively in training until it reaches a minimum threshold.

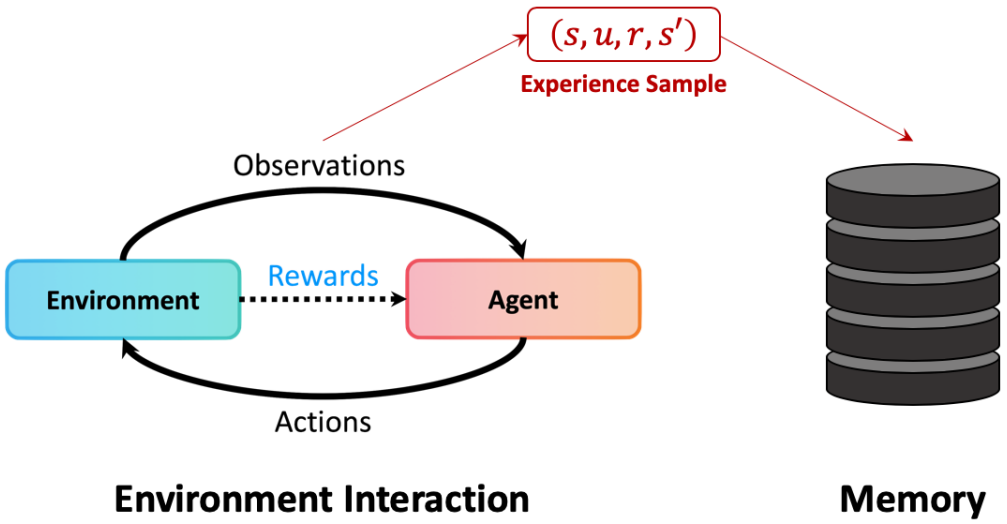


**Fig. 6.1.** Illustration of the experience replay mechanics, where the agent stores its experiences as ($s_t$,$u_t$,$r_t$,$s_{t+1}$) tuples in a memory buffer.

As a last detail, during implementation we can further take advantage of using deep learning as function approximation to accelerate the state-action value evaluation. For efficiency purposes, instead of retrieving a single Q-value for a given state and action, requiring a separate forward pass to compute $Q$ for each action resulting in a cost that scales linearly with the number of actions, we instead use an architecture in which there is a separate output unit for each possible action, and only the state representation in an input to the neural network. This allows us to compute the Q-values for all possible actions in a given state with only a single forward pass through the network (see figure 6.2).

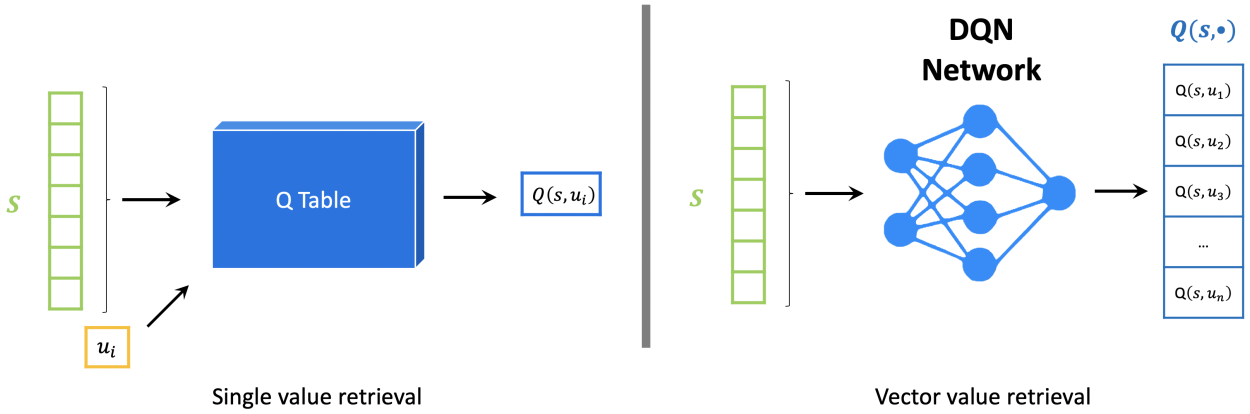


**Fig. 6.2.** Illustration of the classical Q-value retrieval (left), compared to the deep learning approach (right) where the network approximator is a mapping directly from state to actions vector.

## 6.3. Policy-Based Methods and Policy Gradient

Algorithms in this family learn a policy $\pi$ directly, often represented by its own function approximator. Starting from the intuition that good policies generate trajectories that maximize an agent's objective, then they directly create a mapping from an observed state to (probability of) actions. A major convenience of those methods relies in their application generality to discrete, continuous or combined action types, without the need to calculate any value function. Furthermore, even if they usually suffer from higher variance than value-based methods and are more sample-inefficient, by the policy gradient theorem [**72**], they are guaranteed to converge at least to a local optimal policy.

The most fundamental (and simplest) algorithm of this family, representing the basis of almost all other related modern methods, is the *policy gradient* also known as REINFORCE estimator by the RL community. Policy gradient parametrizes the policy directly using a function approximator like a neural network. Combining different ideas, mainly the usage of a stochastic estimator to obtain a derivative estimate of the performance $J$ of a parameterized randomized policy in a MDP, in addition to extended conditional Monte Carlo results in:

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}\left[\sum_{t=0}^{T}\gamma^t r(S_t,U_t)\frac{\nabla_\theta p_\theta(S_0,U_0,\ldots,S_t,U_t)}{p_\theta(S_0,U_0,\ldots,S_t,U_t)}\right]\\ &= \mathbb{E}\left[\sum_{t=0}^{T}\gamma^t r(S_t,U_t)\nabla_\theta \log p_\theta(S_0,U_0,\ldots,S_t,U_t)\right]\\ &= \mathbb{E}\left[\sum_{t=0}^{T}\gamma^t r(S_t,U_t)\sum_{k=0}^{t}\nabla_\theta \log \pi_\theta(U_k|S_k)\right] \quad .\end{aligned}$$

We note that the inner summation could be maintained recursively as a *trace* since it only depends on the past. This is how it is presented in REINFORCE [**81**] and in subsequent work [**73**] by Richard Sutton. Rather than accumulating a sum of gradients, it is more common to switch the order of summations to get:

$$\nabla J(\theta) = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(U_t|S_t\right) \sum_{k=t}^{T} \gamma^{k-t} r(S_k, U_k)\right] \quad .$$

That is, we weight the gradient of the log action probabilities by the return onward from that state. The resulting estimator is then:

$$\widetilde{\nabla}_{\text{REINFORCE}} J(\theta) := \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(u_{i,t}|s_{i,t}) \sum_{k=t}^{T} \gamma^{k-t} r_{i,t} \quad , \tag{6.3.1}$$

where we've assumed that all $N$ trajectories are of length $T$.

## 6.4. Model-Based Methods and Monte Carlo Tree Search

Model-based algorithms either learn a model of an environment's transition dynamics or make use of a known dynamics model. Once equipped with such model, it is then possible to simulate various trajectories virtually without even interacting with the real environment until an action is required. The agent can then analyze the different sequences generated and decide on the best action to take. Such methods are very appealing because they require much less real experience to reach good performances, but are often limited by the difficulty of obtaining a reliable model of the environment. When they work, such methods are often 1 to 2 orders of magnitude more sample-efficient than model-free methods. We refer to algorithms that do not use the environment's transition information as *model-free*.

While most algorithms can simply be augmented with a function approximator to approximate the *state equation* (see section 1.3) of any dynamic system, we choose to introduce an algorithm of historical importance in model-based reinforcement learning, that is the *Monte Carlo Tree Search* (MCTS). In what follows, we develop its basic working principle, before expanding on how it was used in conjunction with deep reinforcement learning and supervised learning to bring a computer Go agent from a weak amateur level in 2005 to a worldwide grandmaster in 2017.

MCTS's execution is triggered as soon as the new state of the agent is known, and is repeatedly applied until an action needs to be outputed to the environment. The core idea of MCTS is to keep a high performing action selection, *while* enjoying a simulated exploratory behavior to see if the value of some actions may not differ from his initial

estimate. More formally, the 4 algorithm steps are as follow, taken from [**72**]:

(1) **Selection**. Starting at the root node, a *tree policy* based on the action values attached to the edges of the tree traverses the tree to select a leaf node.
(2) **Expansion**. On some iterations (depending on details of the application), the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.
(3) **Simulation**. From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy. The result is a Monte Carlo trial with actions selected first by the tree policy and beyond the tree by the rollout policy.
(4) **Backup**. The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS. No values are saved for the states and actions visited by the rollout policy beyond the tree. Figure 6.3 illustrates this by showing a backup from the terminal state of the simulated trajectory directly to the state-action node in the tree where the rollout policy began (though in general, the entire return over the simulated trajectory is backed up to this state-action node).

The chinese game of Go represented a challenge for artificial intelligence researchers for a long time, in part because the number of legal moves allowed per position (approximately 250, compared to 35 in chess for example) creates a very large action space, and in part because of the difficulty to define an adequate position evaluation function considering the rules and objectives of the game. The first generation algorithm deployed by Deepmind, *AlphaGo* [**65**], introduced a novel version of MCTS guided by both a policy and a value function learned through deep reinforcement learning, naming it *asynchronous policy and value MCTS* (APV-MCTS). In this new methodology, the tree expansion strategy was guided by a probability distribution provided by a deep ANN called the *SL-policy network*, trained previously by supervised learning to predict moves contained in a database of nearly 30 million human expert moves. Also in contrast with basic MCTS, the node evaluation was a combination not only of monte carlo rollouts, but also of the value functions $v_\theta$ previously calculated by a value-based deep RL method. Given $s$ a newly added node, its value consequently became:

$$v(s) = (1 - \eta)v_\theta(s) + \eta G \tag{6.4.1}$$

where $G$ was the return of the rollout and $\eta$ controlled the mixing of the values resulting from these two evaluation methods. Finally, the rollout exploratory policy was calculated by a shallow linear network (with an optimal performance vs computation pass ratio) also

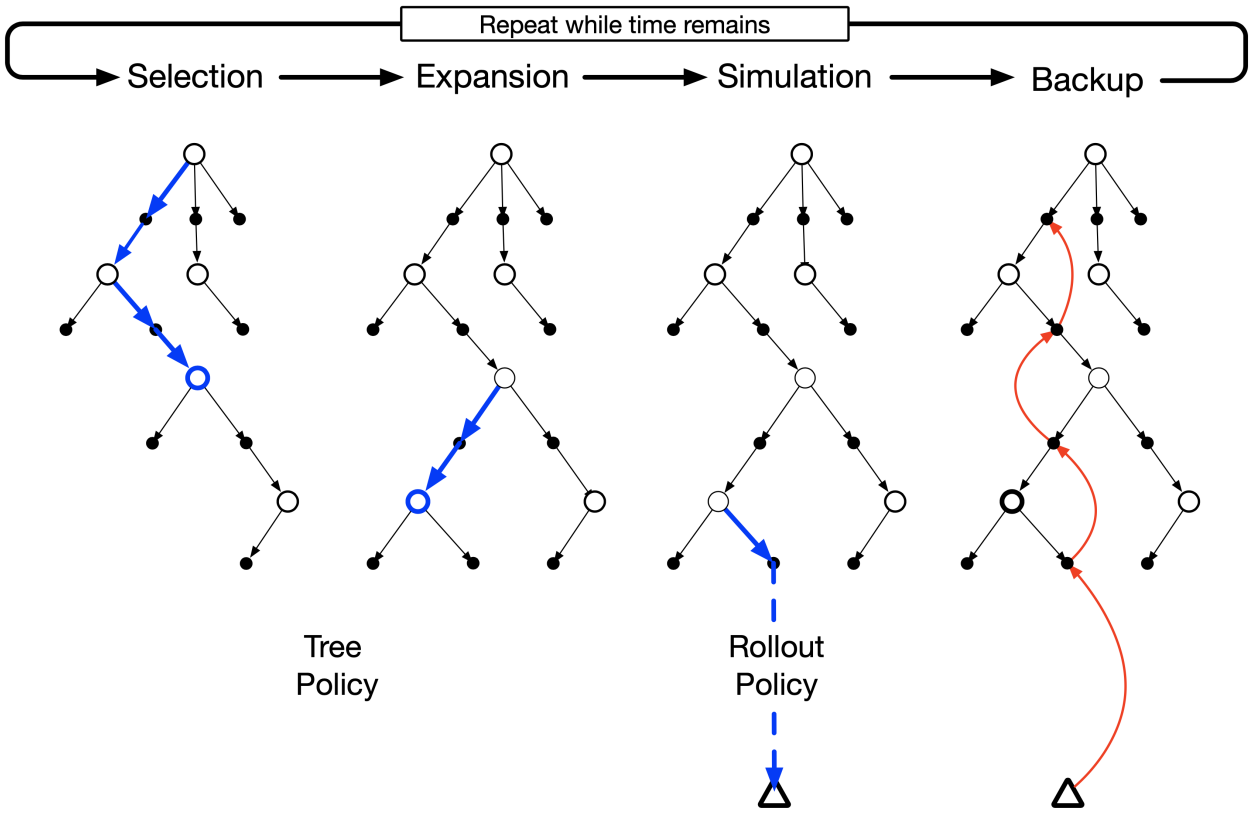


**Fig. 6.3.** Illustration of the four operations in the MCTS algorithm [**72**]: *selection*, *expansion*, *simulation* and *backup*.

previously trained using self adversarial play and human expert moves. The pipeline of this expedited description is illustrated in figure 6.4.

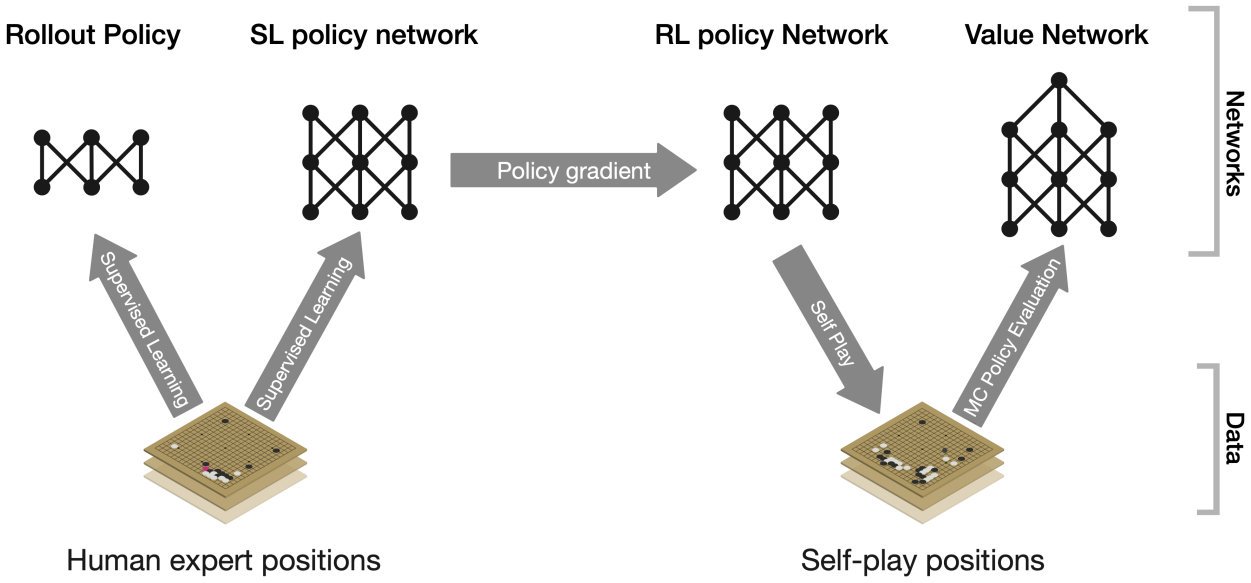


**Fig. 6.4.** AlphaGo pipeline and components [**72**].

Lastly, without going into further details, it is interesting to note that Deepmind achieved new records with their newest *AlphaGo Zero* algorithm [**66**], using no human knowledge or experience samples, except pure deep reinforcement learning.

## 6.5. Hybrid Methods and Developments

We conclude the RL families overview by introducing methods that combine two or more of the three primal families we just presented. Given the strengths and weaknesses of each methodology, it seems natural to combine them to get the best of each. One widely used group of algorithms learns both a policy and a value function. Such algorithms are referred to as *Actor-Critic*, where the actor refers to the policy who *acts*, and the critic to the value function who *critiques* the actions. More formally, starting from the base equation 6.3.1 of the policy gradient, taking crude Monte Carlo updates is problematic because it introduces a high variability in log probabilities and cumulative reward values due to the difference in the trajectories, or is simply not able to update if the cumulative reward is 0. The main idea behind actor-critic methods is to reduce the variance by using a learned *baseline* $b$ instead of the raw return:

$$\nabla J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(u_{i,t}|s_{i,t}) \sum_{k=t}^{T} \gamma^{k-t} r_{i,t} - b(s_{i,t}) \ . \tag{6.5.1}$$

Substracting the return by a state-dependent baseline greatly reduces the variance, and consequently yields smaller gradient updates for the training process. The nature of the chosen baseline leads to different algorithms. Actor-Critic algorithms are an active area of research and have given rise to many interesting developments in recent years, including the following algorithms, to name but a few:

(1) (Asynchronous) Advantage Actor-Critic (A3C and A2C) [**48**].
(2) Trust Region Policy Optimization (TRPO) [**61**].
(3) Proximal Policy Optimization (PPO) [**62**].
(4) Deep Deterministic Policy Gradient (DDPG) [**43**].
(5) Soft Actor-Critic (SAC) [**29**].

To these elements, modern deep learning and different application scenarios allow some pretty powerful models of environment transition dynamics to be added to create all sort of different new algorithms. What we presented is by no mean an exhaustive list, but rather a brief overview of this new field which seems to yield exciting new developments on a regular basis !

## 6.6. Communities and Similarities

We close this chapter by highlighting the different terminologies used by both the statistical learning and the control community to refer to the same concepts. While we noticed this phenomenon on our own during the literature review process of this work, we choose to directly cite page 43 in Dimitri P. Bertsekas' most recent book, *Reinforcement Learning and Optimal Control* [**12**], which expresses the same idea but with much more experienced and in-depth perspective:

> «*There has been intense interest in DP-related approximations in view of their promise to deal with the curse of dimensionality*[3] *and the curse of modeling (a simulator/computer model may be used in place of a mathematical model of the problem). The current state of the subject owes much to an enormously beneficial cross-fertilization of ideas from optimal control (with its traditional emphasis on sequential decision making and formal optimization methodologies) and from artificial intelligence (and its traditional emphasis on learning through observation and experience, heuristic evaluation functions in game-playing programs, and the use of feature-based and other representations). The boundaries between these two fields are now diminished thanks to a deeper understanding of the foundational issues, and the associated methods and core applications. Unfortunately, however, there have been substantial differences in language and emphasis in RL-based discussions (where artificial intelligence-related terminology is used) and DP-based discussions (where optimal control-related terminology is used).* »

Lastly, we also provide Dimitri's list of equivalence terms, which can be useful for readers to understand all the terminology used throughout this thesis and related work from both communities:

(1) **Environment** = System.
(2) **Agent** = Decision maker or controller.
(3) **Action** = Decision or control.
(4) **Reward of a stage** = (Opposite of) Cost of a stage.
(5) **State value** = (Opposite of) Cost starting from a state.
(6) **Value (or reward) function** = (Opposite of) Cost function.
(7) **Action (or state-action) value** = Q-factor (or Q-value, which is also used in RL) of a state-control pair.
(8) **Planning** = Solving a DP problem with a known mathematical model.

[3] See section 4.2.4.

(9) **Learning** = Solving a DP problem without using an explicit mathematical model. Other meanings are also common.
(10) **Self-learning** = Solving a DP problem using some form of policy iteration.
(11) **Prediction** = Policy evaluation.
(12) **Generalized policy iteration** = Optimistic policy iteration.
(13) **State abstraction** = State aggregation.
(14) **Temporal abstraction** = Time aggregation.
(15) **Learning a model** = System identification.
(16) **Episodic task or episode** = Finite-step system trajectory.
(17) **Continuing task** = Infinite-step system trajectory.
(18) **Experience replay** = Reuse of samples in a simulation process.
(19) **Bellman operator** = DP mapping or operator.
(20) **Backup** = Applying the DP operator at some state.
(21) **Sweep** = Applying the DP operator at all states.
(22) **Greedy policy with respect to a cost function** $J$ = Minimizing policy in the DP expression defined by $J$.
(23) **Afterstate** = Post-decision state.
(24) **Ground truth** = Empirical evidence or information provided by direct observation.

# Part 3

# LEVERAGING DEEP REINFORCEMENT LEARNING IN THE SMART GRID ENVIRONMENT

# Chapter 7

# Technicalities

As the first two parts of this work laid the foundations of sequential decision making and statistical learning theories, in this third segment we expose the novel contributions and the nature of the deployed resources. In the present chapter, we begin by disclosing the technicalities of all the achieved implementations and exploited methodologies. Chapter 8 and 9 then build on the introduced content, and illustrates in details results from application on two different instances of the smart grid ecosystem: energy storage units (chapter 8) and smart buildings (chapter 9). Finally, chapter 10 closes by proposing a new application-oriented hybrid mechanism, borrowing content from both the control and artificial intelligence communities, and by discussing the next opportunities and research projects arising from this thesis.

## 7.1. Supervised Learning

This section follows the material introduced in chapters 4 and 5. A reader unfamiliar with either general statistical learning concepts or deep learning should refer to those chapters before proceeding.

### 7.1.1. Data Manipulations

As much as it is impressive to witness the intellectual work behind modern research and challenging to implement it, on the other hand, the amount of resources required to clean and acquire quality data tends to be very often underestimated due to its conceptual simplicity. In fact, from our experience, data manipulations amounts to nearly 50% of all the deployed resources in terms of time for a statistical learning practitioner. Going in the details, we divide the considered data manipulations into four distinct categories as follow: large-scale automated data selection, missing data treatment, traditional preprocessing and data analysis, and time series specific manipulations.

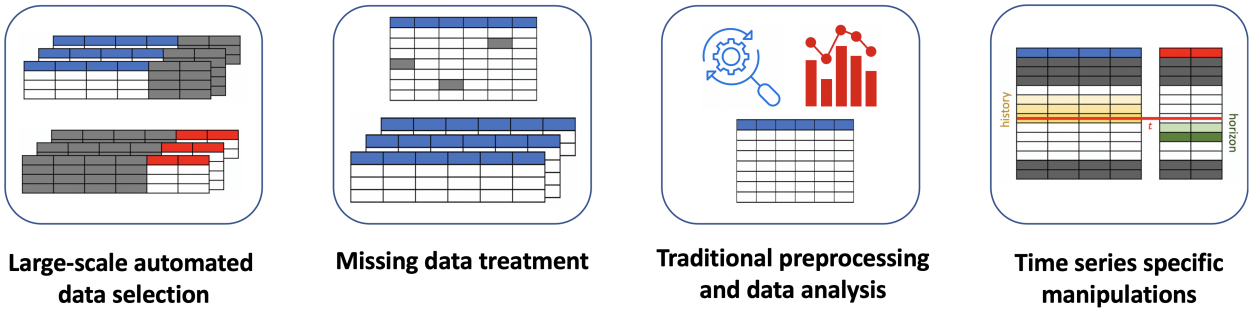


**Fig. 7.1.** List of the main data manipulation types performed on the different application scenarios.

By *large-scale automated data selection*, we refer to two main aspects: (1) acquiring and storing reliable data correlated to our interests, e.g. weather data, API queries (both private and public), time and calendar-related features etc.; (2) playing with *new*[1], big dynamic undocumented datasets, which result from the combined work of multi-disciplinary teams in a continuously-evolving infrastructure. While the former is easily solved with proper documentation and basic implementations, very often the latter is challenging as it requires on-the-go collaboration and coordination from multiple resources, as well as the definition of specific and rigorous conventions.

The *traditional preprocessing and data analysis* category encloses all usual data pre- and post- processing practices, which are easily applied using available material and public libraries. To name but a few of such manipulations, on the preprocessing side: (skewed) data distributions analysis, feature selection, data normalization and standardization, encoding types, and a few others. Post-processing work is also performed, but varies for each application and will be covered subsequently.

For *missing data treatment*, even though it is also part of the traditional duties of a data scientist, some application constraints require fancier methods than normal interpolation or row removal, which is why it is considered a standalone category.

Lastly, the time dependency in the data induces *time series specific manipulations*, which sometimes are inconvenient (removing rows during missing data treatment forces to store separately the subsets of data for example), and some other times are an advantage (when we can use existing methods that were specifically adapted to sequences, for example).

[1]It is undoubtly different to work on data types that were never documented before, unlike NLP or images which are very common and have a rich set of tools and examples available.

### 7.1.2. Deep Learning Training Practices

While various deep learning material was coded from bottom up during academia projects and courses, the reader should be advised that all the presented results implying neural networks herein are realized in combination with the *TensorFlow* library [**2**]. We justify the usage of such library once an appropriate understanding of the deep learning inherent principles is acquired, because native implementation of modern neural network architectures can be very time-consuming, cumbersome, and particularly hard to optimize from a computation speed perspective (which is one of the biggest downsides of deep learning).

In a general attempt to reduce the number of hyperparameters and increase performance on the considered supervised deep learning instances, several procedures are used. First, the training process is set to a very high number of epochs, and interruption can be triggered by only two possible factors: (1) an early stopping criterion monitoring validation loss fires if the value of the loss stopped improving after a certain consecutive number of epochs $\varrho_{ES}$; (2) a (generous) minimal baseline metric value $\beta_{BS}$ is expected after a certain number of epochs $\varrho_{BS}$, or the training is discontinued. This second procedure is put in place to save calculation time and avoid spending resources on architectures lacking promising preliminary results. When the fitting procedure is terminated, the final selected model weights are reverted to the ones that achieved the highest validation performance.

In conjunction to these training interruption criteria, we also apply the strategy of reducing the learning rate (usually by a factor ranging between 2 to 10) upon reaching a generalization error plateau, for some patience parameter $\varrho_{LR}$. We further introduce the condition $n\ \varrho_{LR} < \varrho_{ES}$, allowing the training procedure to reduce the learning rate at least $n$ consecutive times before triggering the early stopping training termination. This way, early stopping is guaranteed to be prompted only after several learning rate reductions, and we can be almost assured that no further improvement was possible by reducing the learning rate, and that we likely terminated in the lowest point of a local convex shape on the error surface.

Traditional hyper-parameters grid search is not well suited for deep learning, in part because of the important training time, and in part because of the initialization process which may result in noisy samples. Considering this aspect, we rely on Bayesian optimization (see section 4.5) and consider the following problem:

$$x^{\star} = \arg\min_{x} f(x) \ , \tag{7.1.1}$$

where $f$ is our training resulting validation performance for a set of specific hyper-parameters. Furthermore, $f$ is expensive to evaluate, has no known closed form or gradients, and has noisy evaluations of $y = f(x)$. The Bayesian optimization loop is made of two steps: (1) building a probabilistic model for the objective $f$, by integrating out all possible true functions using Gaussian process regression; (2) Optimizing a cheap acquisition/utility function $u$ based on the posterior distribution for sampling the next point. The next observation is then sampled and the process is repeated. For more details on the implementation, an interested reader can consult the *scikit-optimize* library documentation [**64**].

### 7.1.3. Attention in Deep Learning

*Attention* is an important contribution to the field of deep learning that was first introduced in 2015 by *Bahdanau et al* [**6**] in the context of neural machine translation (NMT). The main idea behind the original Attention was to solve the *long-range dependency problem* of Seq2Seq models, that were shown to have an important performance degradation scaling with sequence input length [**70**]. To put it simply, Attention is a weighted average of $J$ values $h_j$:

$$c = \sum_{j=1}^{J} \alpha_j h_j \ , \tag{7.1.2}$$

where $\sum \alpha_j = 1$. The product $\alpha_j h_j$ is referred to as the *alignment vector*, and can be interpreted as the "contribution" of each value $h_j$. In its initial form, which is commonly referred today as *Bahdanau Attention* or *Additive Attention*, the mechanism uses a MLP to output *alignment scores*[2] $e_j$ that are fed in a softmax activation to produce $\alpha$:

$$\alpha_j = \frac{\exp(e_j)}{\sum_{k=1}^{J} \exp(e_k)} \ . \tag{7.1.3}$$

In the original paper, the authors sum the alignment vectors like in equation 7.1.2 to produce a specific context vector $c_i$ for each output of the network ($i$ indexes the $i$th output of the decoder), using the encoder's hidden states and the previous state of the decoder. The score function is thus of the form:

$$e_{ij} = a(s_{i-1}, h_j) \tag{7.1.4}$$

where $s_{i-1}$ is the decoder's RNN hidden state just before emitting $y_i$, $h_j$ the $j$-th annotation of the input sequence of length $J$ and $a$ a (non-linear) activation function. *Luong's et al.* [**44**] later simplified and generalized the attention concept to a broader definition, where the Additive Attention is simply a characterization of this new formalism.

[2]Different score functions can be used, as shown in figure 7.2.

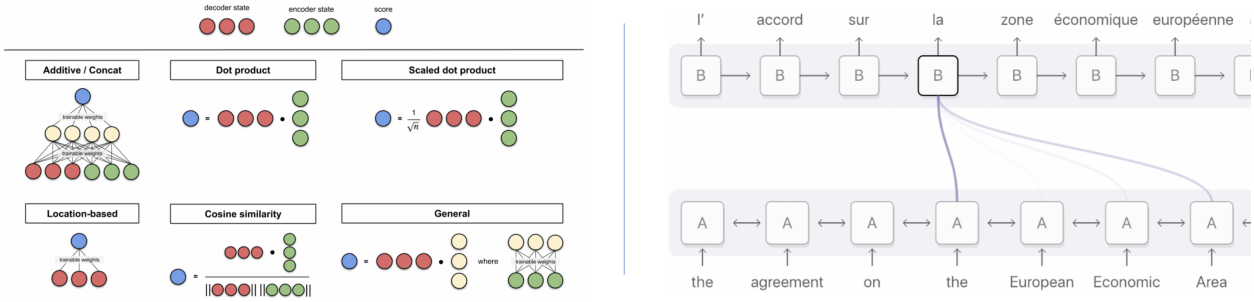


**Fig. 7.2.** Illustration of: (left) the different score functions that can be used with Attention; (right) the *alignment* concept in a translation from English to French, for the specific word "la" across the input sequence [**50**].

Given encoder and decoder sequences of respective length $m$ and $n$, the aforementioned schematic requires to pass $mn$ times through the network to acquire all the attention scores. *Vaswani et al.* in their iconic *Attention is all you need* [**77**] paper propose to first project $s$ and $h$ onto a common space, then to choose a similarity measure as the attention score, like the dot product:

$$e_{ij} = f(s_i)g(h_j)^T \quad . \tag{7.1.5}$$

In this case, $g(h_j)$ and $f(s_i)$ only need to be calculated $m$ and $n$ times, respectively, to get the projected vectors which can then be used to compute efficiently the alignment scores.

It is interesting to compare the previous development to a retrieval process. That is, multiplying $h_j$ with $\alpha_j$ can be seen as a "proportional retrieval", or directly retrieving a specific element $h_j$ if we restrict $\alpha$ to be a one-hot vector. In a similar fashion, the two projections $f(s_i)$ and $g(h_j)$ can be interpreted as a *query* (for the decoder) and as *keys* (for the encoder). *Vaswani et al.* highlighted this analogy, which is now commonly used as we refer to Attention as a general mapping or retrieval process taking one or multiple **keys** $K$, **queries** $Q$ and **values** $V$:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad , \tag{7.1.6}$$

where $d_k$ is the dimensionality of the queries and keys. The research group also introduced a novel *Transformer* architecture, showing translation performances exceeding all the related previous literature results, and relying only on a combination of different Attention mechanisms.

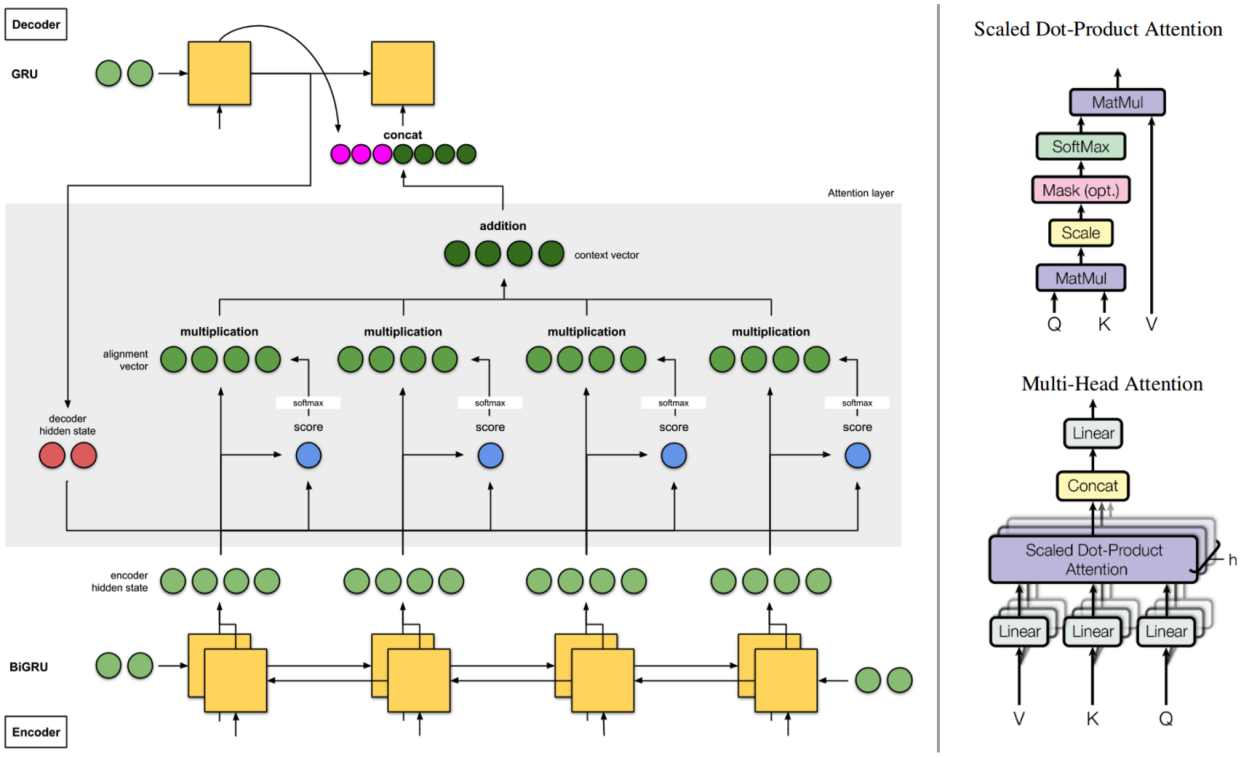


**Fig. 7.3.** Illustration of: (left) NMT from *Bahdanau et al* [**6**] - Attention Seq2Seq architecture with bidirectional GRUs used by the authors [**37**]; (right) Scaled Dot-Product and Multi-Head Attention mechanisms introduced by *Vaswani et al.* [**77**].

*Local Attention* and *Global Attention* are important concepts derived from the general application of Attention. While all inputs are considered in the Global Attention computation, which results in an important computational cost, Local Attention tries to focus only on a subset of elements. That is, using a position $p_t$ in the input sequence, the system considers only the items in a window of the range $[p_t - D, p_t + D]$, where $D$ is a chosen hyper-parameter. This position can either be learned using a mechanism ending with a sigmoid activation, which returns the relative position in the sequence length (the value 0 representing the beginning, 1 the end), or can heuristically be set relative to the current time sequence input $t$. The former procedure is referred to as *predictive alignment*, and the latter as *monotonic alignment*.

### 7.1.4. Deep Learning Architectures

With proper learning practices set up and all the necessary material presented, we now introduce the main neural network architecture families examined in the context of this work (see figure 7.4); which are focused and adapted for multivariate time series data types

[**24**]:

(1) **Multi-Layer Perceptron**: archetype feed-forward neural network made of consecutive fully-connected layers (see section 5.1).

(2) **Convolutional Neural Network**: consecutive alternance of convolution, batch normalization and pooling operations (see section 5.2), followed by one or multiple dense layers. While there seems to be a polemic whether batch normalization should be applied before or after activation, empirical results motivated us to use the first option.

(3) **Recurrent Neural Network**: stacked gated recurrent layers (see section 5.3), can output directly a sequence (output at each input of the sequence) or a vector (final output after receiving the whole sequence). The reader should note that all considered recurrent layers are gated (GRU, LSTM) and may also be bidirectional.

(4) **Residual Network** [**31**] (ResNet): following the observed performance gains of deeper CNNs, these architectures were introduced to help the gradient flow in many-layered settings. ResNets are made of *residual blocks*, that in its simplest implementation adds the identity value of its input to its output (which is usually the result of one or more convolutional layer).

(5) **Convolutional Recurrent Neural Network** (CNN-RNN): originally referred to as *Long-term Recurrent Convolutional Networks* (LRCN) [**19**], such structure is a combination of convolutional layers applied repeatedly on (subsequences of) the original sequence, followed by recurrent layers. This type of network allows the convolution to extract time translation-invariant (or other specific transformation) features, which are then processed in the following recurrent layers. If subsequences of the original sequence are obtained using a sliding window, then fed to a CNN, literature typically uses the term *multi-scale* convolutional neural networks (MCNN) [**18**]. It is worth noting that a global average pooling or other similar down-sampling layers must be added in this case, to make the MCNN output match the correct tensor dimensionality expected by an RNN.

(6) **Wide and Deep architectures** [**13**]: concept combining both the output of various stacked deep layers, and the direct linear transform of the original input, to allow the final network's layer to rely on a combination of inputs and higher-level features.

(7) **(Attention) Sequence to Sequence Encoder-Decoder** [**14**] [**71**]: Sequence to sequence (or encoder-decoder) architecture consists in an encoder part synthesizing the input sequences into a fixed-length *context* or *thought* vector, which is then used as input for the decoder that outputs another sequence (see section 5.3.2). Modern architectures like the ones described in the previous subsection also typically add an

attention component, which "soft-searches" for sequence parts specifically useful to predict each output.

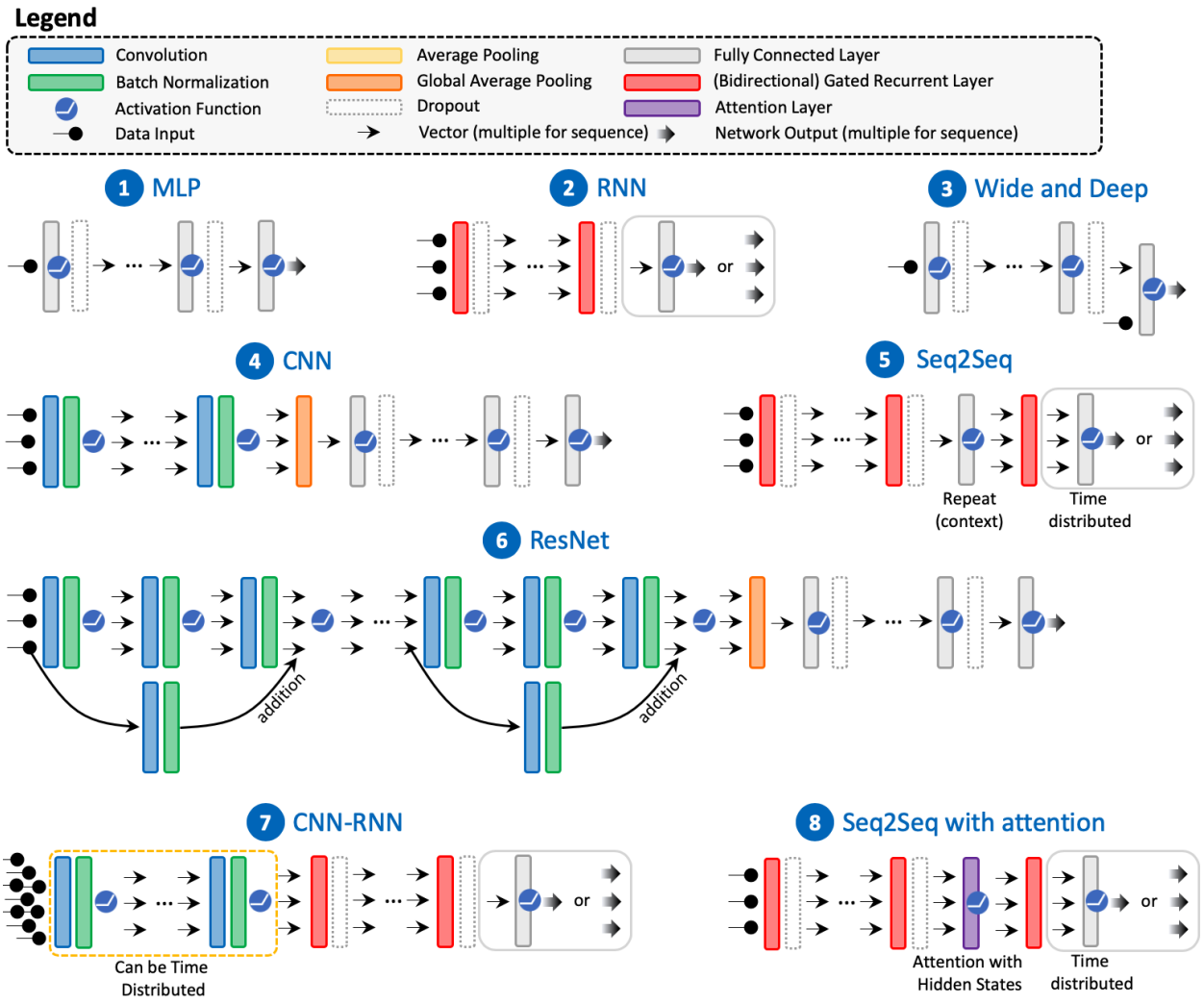


**Fig. 7.4.** Sequence-oriented deep learning architectures families explored in the context of this work. Activation functions are not explicitly shown for (gated) recurrent layers, due to their more complex internal pattern. The reader should note that this figure is purely illustrative, and may not necessarily be true from an implementation or mathematical point of view.

While we tried to break down the architectures into different "types" or families, it is important to understand that these do not constitute, in any mean, an exhaustive list or an officially recognized nomenclature. Any kind of layer or mathematical operation could be added in any part of a neural network in the above list, and it would result in a new entity whose category affiliation could be debated.

Leveraging this reality and flexibility as an opportunity, as it is now common practice in the deep learning community, we combine and output different data types, or data with different temporal granularities into the same deep learning model by using tensor copying, splitting and merging operations (addition, concatenation, point-wise, cross-product ...) with different layer types. As we will see in one of the application scenarios, such approach can even be used to perform missing data inference, at the cost of higher computational resources, by applying encoder-decoder pre-training to reconstruct the occasionally missing inputs.

### 7.1.5. Classical Machine Learning Algorithms

Finally, other classical machine learning algorithms were also used in application, to name but a few: *support vector machines* (SVM and SVC), *K-Nearest Neighbors* (KNN), *Decision Trees*, and *Random Forests*. However, their sporadic usage either as a baseline or as a quick utility made us consider them a simple supervised learning black box model, using the *Scikit-Learn* library [**52**].

## 7.2. Reinforcement Learning and Control

Many popular libraries are readily accessible for tasks ranging from basic data manipulations to deep learning (as we saw earlier). While a few interesting dynamic programming packages are available, they are not friendly at all to the structure traditional deep reinforcement learning is implemented on. On the other hand, DRL being in its infancy at the time of this writing, the only interesting research package we encounter is *openAI gym*, which despite being a great initiative from the community to create benchmarks and standardized "datasets", is less adapted to the more complex applied instances with real data we consider. With this in mind, while we ensure full compatibility of our controllers and environments with both external libraries, to validate and compare control and DRL algorithms, we build our own infrastructure for simulation and control from the bottom up. This mainly includes graphs (based on dictionaries and adjacency matrices), classical control algorithms, deep reinforcement learning algorithms, environment simulation and combination with real data. The idea is on one hand to use this as a didactic opportunity, and on the other to have the required flexibility to combine all the necessary ingredients together.

Despite several algorithms implementation, our primary DRL application focus, and the majority of available resources are spent on the deep Q-Network. Our choice is motivated in part because of its off-policy (see section 6.1) and data-efficient nature, which allows it to learn directly on external collected experience samples (something very common in application), and partly because the nature of the quantity it approximates can be quite

valuable and manipulated in a lot of different ways for various purposes, as we will see in the upcoming sections.

### 7.2.1. Improved Deep Q-Network

We implement the classical DQN algorithm (see section 6.2.1) as per [**47**], and extend its framework in several different ways. First, it was shown in the literature that Q-learning has a tendency to overestimate action values due to its optimistic maximization operation [**75**]. Using the second target network to compute the state-action values and the original Q-network to choose the action is an adapted double Q-learning implementation [**72**], which offers the best computation vs result compromise to solve the overestimation problem, known as the *Double Deep Q-Network* algorithm [**76**]. Another interesting implementation is the *Dueling* variant of the DDQN (D3QN) [**79**], which involves separating the Q-value into its advantage $A$ and state-value $V$ components (see section 6.1). This process can be done directly in the deep neural network through a special aggregation layer (see figure 7.5). By decoupling its Q-value prediction, the network can learn which states are (or are not) valuable without having to learn the effect of each action at each state and has proven to yield a better generalization performance. Also, instead of using a deterministic greedy policy with probability $\epsilon$ of taking a random action, like the one in [**47**], we make profit of having access to the advantage value and use it as a random policy with probability $1 - \epsilon$, because $A$ has the same properties as a probability distribution. Doing so is typically referred to as using a *Boltzmann* policy in the DRL community, and bolsters an appropriate exploratory behavior in the long term, while still allowing convergence towards an optimal random policy if needs be.

From a function approximation perspective, we leverage the previously developed deep learning material, and re-use all the aforementioned architectures for function approximation performance. Secondly, we perform in a statistical learning way, the equivalent of what is known as *state augmentation* in dynamic programming. That is, at the expense of additional computing cost, we increase the amount of information contained in the state input and map directly observation sequences $\{o_{t-H},...,o_{t-1},o_t\}$ to $Q$-values[3], where $H$ represents the *history* considered by the agent. Doing so allows the decision maker to increase the amount of statistical relationships, and has proven particularly powerful in the partially observed setting, which is almost always the case in real applications.

With a practical application point of view, we further leverage the flexibility of the deep learning infrastructure by considering not only a single $Q$-value vector for all the possible

[3]To preserve the Markovian assumption we introduced in the early chapters, we informally consider this sequence as the state, without any loss of generalization and to avoid unnecessary corrections.

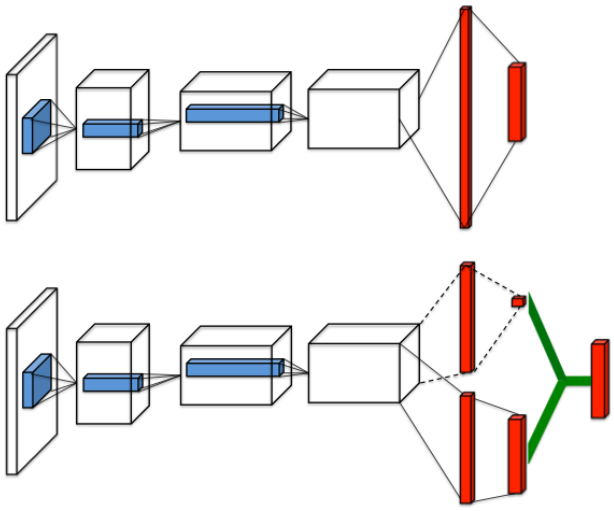

**Fig. 7.5.** Illustration of the Dueling Deep Q-Network principle [**79**]: the Q-function estimate is separated into advantage $A$ (bottom layer) and state-value $V$ stream (top, unitary layer) components inside the network, before being aggregated at the output.

actions, but rather parallel $Q$-value output streams for every control system. This can be seen as a simplistic multi-agent framework adaptation, as it allows on one part an important action space factorization, and on another the possibility to provide system-specific actions and granularities for each controller. It is also computationally-efficient because it requires only a single forward pass calculation, while still benefiting from shared parameters which offer a global generalization and optimization scheme[4].

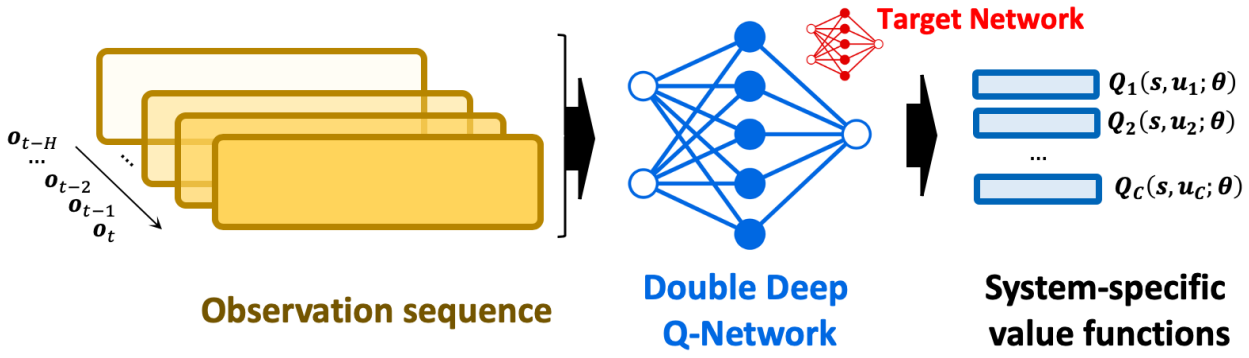


**Fig. 7.6.** Upgraded Double Deep Q-Network using time series-adapted deep learning architectures to map a sequence of observations $\{o_{t-H}, ..., o_{t-1}, o_t\}$ to individual parallel $Q$-value vectors, one for each of the $C$ control systems.

Lastly, as a combined result of community discussions and trial and error, we modified a few settings over their traditional or default values. First, we used an *He initialization*

[4]And also because we consider a centralized information setting, where the individual and combined rewards are accessible to all agents. More on this in the chapter 9.

[**32**] also known as *variance scaling* instead of the traditional *Xavier* initialization (see section 5.5.2). While both methods aim at making the variance of a layer equal to the variance of its inputs, empirical evidence seem to show that *He* is more approriate for DQN. Second, in a similar fashion to traditional ANN training, we execute scheduled learning rate reduction, starting from a step size of $1 \times 10^{-3}$ and ending at a step size of $1 \times 10^{-4}$ at 80% of the training with *Adam*'s optimizer [**38**]. We also clip the norm of the gradient to be between $-1$ and 1, to avoid updates that would be too drastic and destabilize the learning process. Finally, we initialize the exploration rate $\epsilon$ to 1 for training, and perform a linear annihilation to reach a floor value of $\epsilon = 0.05$ at 80% of the process.

Despite a high potential for performance exploration and tuning, we purposely try to fix and reduce to a minimum the number of hyper-parameters. Our objective by doing so, is to test application robustness and flexibility to a broad range of different instances, while reducing the associated computational burden. This is also useful to demonstrate deployment autonomy to third parties, as no post-implementation interventions are required.

### 7.2.2. Policy Gradient and Advantage Actor-Critic

As an archetype algorithm of the policy-based reinforcement learning families, and of control theory in general, we also proceed to test the classical policy gradient (REINFORCE) algorithm [**81**] (see section 6.3). Sadly, application on the considered instances returned poor performances and longer training times compared to the other available DRL algorithms. But such result is not surprising, as the sample-inefficiency problem of this method is known in the literature, and the relatively small action spaces considered in our instances suggest value-based methods are more adapted in the present context.

From the policy gradient and the DQN implementations, we have all the tools to build the A2C and A3C algorithms [**48**] (see section 6.5), which is a hybrid algorithm of type *actor-critic*. We recall that actor-critic methods combine a separate architecture to explicitly represent the policy independent of the value function. The former is referred to as the actor, and the latter as the critic.

In the original paper [**48**], the critic is a deep neural network approximating the state-value function $V$. The cost function for the policy function results then in:

$$f_\pi(\theta) = \log \pi(u_t|s_t;\theta)\Big(R_t - V(s_t;\theta_t)\Big) + \beta H(\pi(s_t;\theta)) \ , \tag{7.2.1}$$

where $\theta_t$ are the values of the parameters $\theta$ at time $t$, $R_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k};\theta_t)$ is the estimated discounted reward in the time interval from $t$ to $t+k$ ($k$ is upper-bounded

by a constant $t_{max}$), and $H(\pi(s_t;\theta))$ is an entropy term used to favor exploration during the training process. The cost function for the estimated value function then becomes:

$$f_v(\theta) = (R_t - V(s_t;\theta))^2 \ . \tag{7.2.2}$$

Training is performed by collecting the gradient $\nabla\theta$ from both of the cost functions, and applying an incremental gradient descent algorithm in a centralized fashion. One of the advantages of this algorithm is that multiple agents can be performed in parallel without any GPU usage. If the central system waits for all gradients to perform the update, we refer to the algorithm as A2C, while if the update is performed as soon as the gradient of one agent is received, then the algorithm is known as *Asynchronous* A2C (A3C). Once training is performed, the server then sends back an updated version of the parameters to the agents to guarantee they share a common policy.

In our case, we reuse the available D3QN and make profit of our simultaneous access to all $Q$, $V$ and $A$ values to use them as a baseline for the actor, leading to the following:

$$\nabla J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(u_{i,t}|s_{i,t})A(s_{i,t},u_{i,t}) \quad \{\text{A2C or A3C}\} \ , \tag{7.2.3}$$

$$\nabla J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(u_{i,t}|s_{i,t})Q(s_{i,t},u_{i,t}) \quad \{\text{Q Actor-Critic}\} \ . \tag{7.2.4}$$

While in [**48**] the authors prefer to use the following equality to calculate the advantage $A$:

$$A(s_t,u_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \ , \tag{7.2.5}$$

we simply preferred to use our value from the D3QN, which we found less cumbersome and more computationaly efficient.

Lastly, just like for the DQN implementations, we used similar settings for the deep learning: the $He$ initialization, the scheduled learning rate and gradient clipping. We also fixed our hyper-parameters to test robustness with minimum intervention.

# Chapter 8

# Smart Grid and Energy Storage

This chapter presents the first set of considered application instances, using the smart grid as environment along with its actors, and deploying the methodology and tools described up to this point. The reader should be aware that the content that follows was chronologically realized in the early stages of this work, and should consequently be viewed as a development and validation phase that is now obsolete at the time of writing. The material was developed in collaboration with the industrial partner *Sigma Energy Storage*, and private data has been replaced with random sampling from a similar distribution when necessary. Financial problems forced the company to cease all its activities mid way through the 2019 year, which results in the fact that the projects described below are no longer active or under development.

## 8.1. Smart Grid

As the first power grids were originally designed to be centralized unidirectional systems of electric power transmission and distribution [**45**], in what we consider today's modern Smart Grids, power generation is shifting from a centralized to a distributed approach with four main actors interacting and communicating continuously together: (1) Energy generation, transmission and distribution resources; (2) Markets; (3) Customers and service providers; (4) Operations control. More formally, as defined by the European Union Comission Task Force for Smart Grids [**25**]:

> «*A Smart Grid is an electricity network that can cost efficiently integrate the behaviour and actions of all users connected to it - generators, consumers and those that both - in order to ensure economically efficient, sustainable power system with low losses and high levels of quality and security of supply and safety. A smart grid employs innovative products and services together with intelligent monitoring, control, communication, and self-healing technologies [...].*»

The dynamic optimization requirement of grid operations and resources, makes this optimal control problem a perfect candidate for dynamic programming and reinforcement learning approaches. Furthermore, the changes resulting from the optimization of all the implied agents will allow improvement in certain large-scale problems related to the production and distribution of energy on the electricity grid, a decrease in the individual bill for consumers, and more importantly, a direct benefit for the environment and sustainable development.

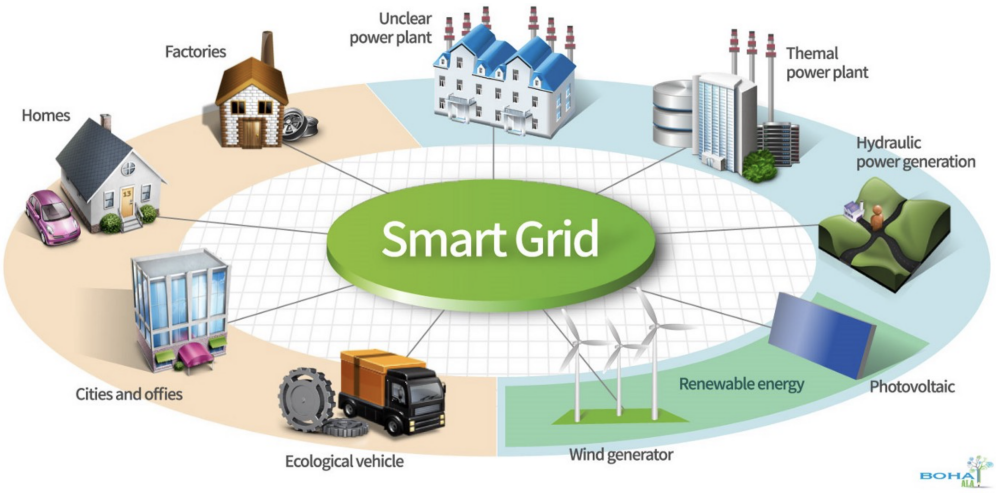


**Fig. 8.1.** Conceptual illustration of the smart grid and its inherent actors [**74**].

## 8.2. Energy Storage

### 8.2.1. Description

As the diversity and efficiency of energy resources continue to increase, the large-scale conversion and storage of energy, whether in chemical, thermal, mechanical, electrical or inertial form, represents one of the major industrial rivalries and technological challenges of our modern times. An energy storage, often called an *accumulator* or a *battery*, is a device capturing[1] energy produced at one time to use it at a later time. Its main application utilities are:

(1) **Load leveling and peak shaving**: store energy during low demand and deliver during high demand.

(2) **Intermittent renewable integration**: avoid curtailment and store intermittent energy for later delivery.

(3) **Electricity arbitraging**: store energy when grid prices are low, then sell when they are high.

[1]The capturing process actually involves *converting* the energy into another form, following one of the must fundamental laws of physics.

(4) **Network infrastructure and resiliency**: increased reliability, frequency regulation, black starts, etc.

### 8.2.2. Modeling and Optimization

Our general goal consequently is to optimize the behavior of any agent presenting an energy storage capacity, with respect to its objective, regardless of its nature or technology. To do so, we define and characterize the following general properties of an energy storage system:

- Total energy storage capacity $C \in \mathbb{R}^{\geq 0}$.
- Default charging efficiency $\eta_i \in [0,1]$, arising from energy losses in the charging conversion process.
- Default discharging efficiency $\eta_o \in [0,1]$, identical to above but for the discharching process.
- Temporal energy decay following $E \propto E_0 e^{-\lambda t}$, where $E$ is the current internal energy, $E_0$ the initial energy stored, $\lambda \in \mathbb{R}^{\geq 0}$ a decay parameter, and $t$ indexes time. This represents the system's energy "leakage", losing charge over time if it is not discharged rapidly. It is interesting to note that this mechanism, just like the charging and discharging actions, further breaks the Markovian property of the system as the new energy values depend of their historical counterparts.
- Minimum input power $p_{\text{in}}^{\min} \in \mathbb{R}^{\geq 0}$.
- Minimum output power $p_{\text{out}}^{\min} \in \mathbb{R}^{\geq 0}$.
- Maximum input power $p_{\text{in}}^{\max} \in \mathbb{R}^{\geq 0}$.
- Maximum output power $p_{\text{out}}^{\max} \in \mathbb{R}^{\geq 0}$.
- Charging ramp up time $T_r^{\text{ch}} \in \mathbb{N}^{\geq 0}$, i.e. preparation time before the system can actually proceed to charge. Defaults to 0 unless specified.
- Discharging ramp up time $T_r^{\text{ch}} \in \mathbb{N}^{\geq 0}$, i.e. preparation time before the system can actually proceed to discharge. Defaults to 0 unless specified.

We discretize the actions as an odd number, where one action (typically the index median) corresponds to the "do nothing" control, while the others represent an equally spaced fraction of the maximum charging and discharging powers[2] $p_{\text{in}}^{\max}$ and $p_{\text{out}}^{\max}$. Denoting by $p_{\text{in}}$ the combined power input in the energy storage during the charging process, the system's energy and environment's energy difference are given by:

[2]It is important to note that the physical property of some systems also establish a floor value greater than 0 for $p_{\text{in}}^{\min}$ and $p_{\text{out}}^{\min}$.

$$E_{\text{storage}}^{(t+1)} = E_{\text{storage}}^{(t)} e^{-\lambda} + p_{\text{in}} \Delta t \ , \tag{8.2.1}$$

$$\Delta E_{\text{env}}^{(t+1)} = - \, p_{\text{in}} \Delta t \ \eta_{\text{in}} \ , \tag{8.2.2}$$

where $\Delta t$ represents the chosen time step increments for the simulation. Inversely, taking $p_{\text{out}}$ as the total power output from the battery in the discharge process yields:

$$E_{\text{storage}}^{(t+1)} = E_{\text{storage}}^{(t)} e^{-\lambda} - p_{\text{out}} \Delta t \ , \tag{8.2.3}$$

$$\Delta E_{\text{env}}^{(t+1)} = p_{\text{out}} \Delta t \ \eta_{\text{out}} \ . \tag{8.2.4}$$

The four equations above can of course be manipulated to oblige certain constraints, e.g. to derive the value of $p_{\text{in}}$ or $p_{\text{out}}$ if a specific power input or output is expected in the environment from the storage device. Finally, we also add the restriction $0 \leq E_{\text{storage}}^{(t+1)} \leq C$ to account for the finite physical storing capabilities of the accumulator. While the units vary with the scale of the application instances (from up to 3 orders of magnitude), we favour the usage of the international (SI) units in simulation. The aforementioned development defines the environment transition dynamics of our model.

The DQN and A2C algorithms, as described in Chapter 7, are used to drive the control. In each application scenario, the best result among the two algorithms is shown, and an MLP containing one hidden layer of 256 units with ReLU was used as function approximator in both cases.

## 8.3. Literature Review

A wide range of problems in energy systems require making decisions in the presence of some form of uncertainty. While many different literature resources tackle the energy modeling problem, a fundamental synthesizing work has been realized by *Powel et al.* to introduce a straightforward canonical model, which presents four fundamental classes of policies derived from competing strategies proposed by control theory, dynamic programming, stochastic programming and robust optimization. In the first part of their paper [**54**], the authors highlight the importance of separating the modeling of a problem, with the design of policies to solve it. In the second part [**55**], Powell and Meisel further illustrate the application of the proposed modeling framework by considering a typical energy storage problem, and by providing additional discussion behind subtle concepts such as the state variable construction. Lastly, the authors also illustrate that each of the

fours aforementioned classes of policies may be best depending on the problem's characteristics, and introduces a combined approach to combine strengths of multiple policy classes.

The novel content presented in this thesis builds on recent subsequent work introduced by the same author, in the perspective of developing a unified framework for stochastic optimization [**53**], and extends the presented material on different concrete application scenarios.

## 8.4. Validation and Toy Examples

### 8.4.1. Renewable Energy Sources

Even if individual components are *unit tested*, i.e. verified using specific individual tests for algorithms, optimization, control and simulation; the complete interacting process pipeline needs to be validated using toy examples of typical instances, where the control performance can be cross-checked with known or analytical results.

The first toy instance, also acting as a proof of concept (POC), demonstrates the DQN's capacity to simply minimize energy consumption when connected to different typical renewable energy sources. The considered simplified distributions are depicted in figure 8.2, and control sequences can be confirmed to reach an optimal (or at least satisfying $\epsilon$-optimal solution) using the DP algorithm, given a sufficiently coarse granularity (to reduce the possible state and action space).

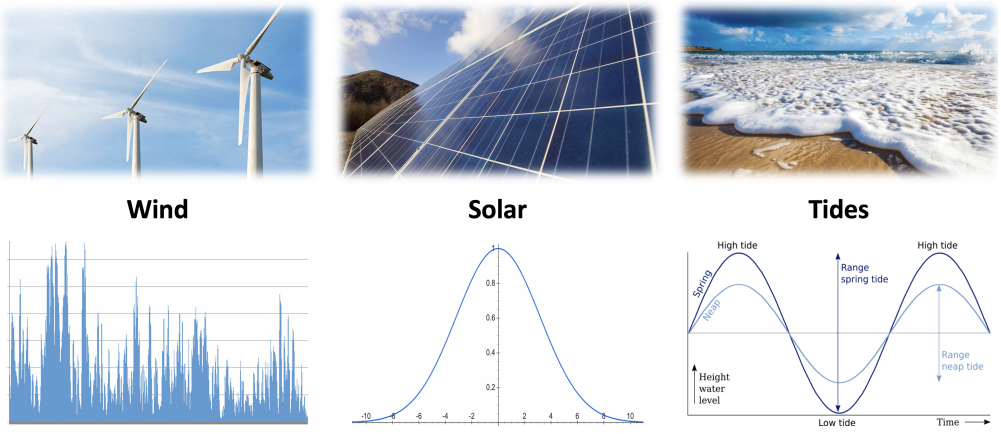


**Fig. 8.2.** Illustration of the typical profile of wind, solar and tidal renewable energy sources.

### 8.4.2. Household Consumption and Electric Vehicle

The last validation instance is a traditional household connected to the grid, where the controller manages the electric car's charging and discharging schedule. In this

simulation, we consider a 100% efficient 100 kWh capacity electric car, and a household energy consumption drawn from a normal distribution where the mean over time is given by the traditional *duck curve* value, and has a standard deviation of 0.3 kW. The car leaves randomly between 6 and 9 AM, and returns between 3 and 7 PM. We require from the system to be at least at 80% of its maximum charge upon departure, and at least at 40% before midnight. The energy price from the grid is fixed and varies with the time of the day, also according to the duck curve distribution. One of the results of the simulation and the control characteristics are detailed in the figure 8.3. It is worth noting that the state includes the time $t$ and cyclic components $\sin(2\pi t/24)$ and $\cos(2\pi t/24)$.

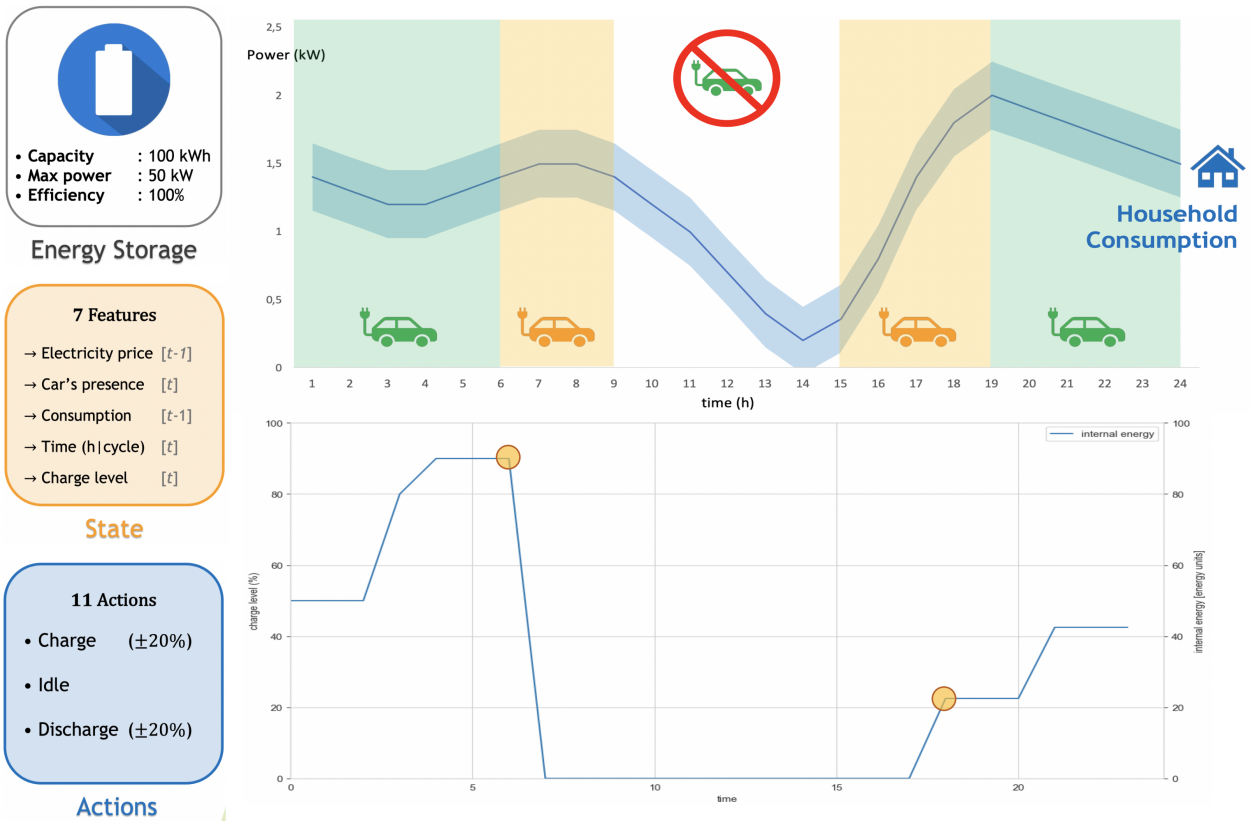


**Fig. 8.3.** Toy example of a household with electrical vehicle, where the DQN controller manages an electric car's charging and discharging schedule. The left squares show the energy storage's characteristics, along with the considered state and actions; the top right image illustrates the simulation's details; and the bottom right image shows the internal energy evolution of the electric car over time (considered null while it is away), where the orange circles depict the departure and arrival time on that specific day.

Results show a proper behavior from the DQN controller, which fulfils all constraints while waiting for electricity prices to lower in the morning and evening before charging, thus minimizing the overall cost and energy consumed. While it did not explicitly perform peak

shaving on the specific illustrated day, discharging to reduce costly energy usage from the grid has been observed on days with earlier arrivals or higher battery levels.

## 8.5. Intermittent Renewable Integration

The intermittent renewable integration simulation resembles the electric car scenario, except for its larger scale and permanent presence. Since it relies on private customer data, we replace the real consumption by using samples drawn from a normal distribution with a fixed mean of 20 kW and a standard deviation of 2 kW. While simplistic at first sight, the resulting samples turn out pretty similar to their original counterparts, given the continuous nature of the industrial process considered. We further simulate the effect of solar panels contribution from a Gaussian distribution with standard deviation of 2 kW, but with a mean varying over time to reach a maximum of 20 kW at the sun's zenith, at 12PM (see figure 8.4). All the other elements are identical to the previous section, and the car's presence feature is replaced by the solar generation at the previous time step.

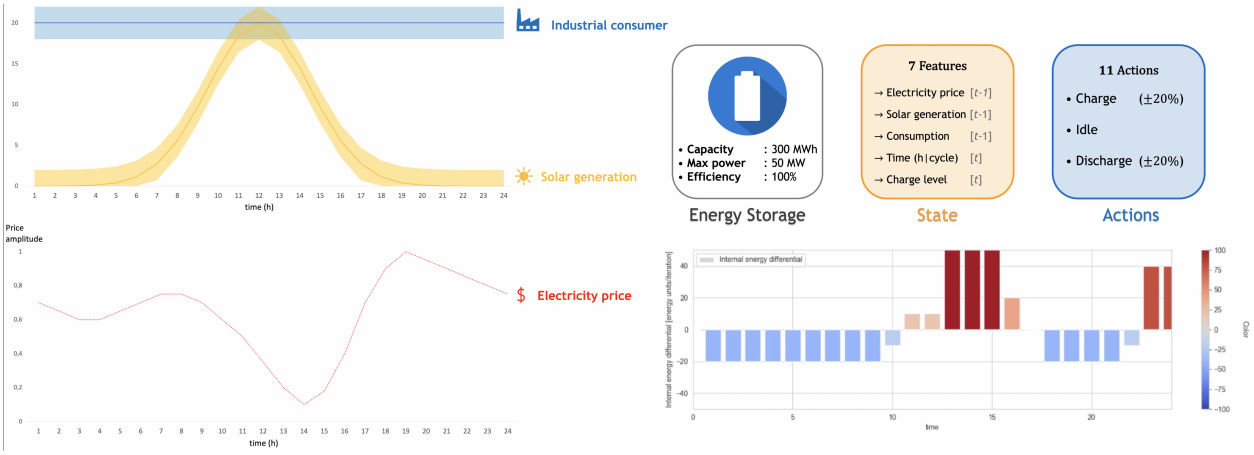


**Fig. 8.4.** Intermittent renewable integration scenario, where the A2C controller manages an energy storage's operations interacting with solar panels and an industrial customer. The top right squares show the energy storage's characteristics, along with the considered state and actions; the left image illustrates the simulation's details; and the bottom right image shows the hourly controls on a specific day.

This example presents a particular interest from an optimization perspective, because it requires the control system to exit a local optimum during training: instead of simply discharging then remaining idle, which results in a tempting immediate reward, the real global optimum must be attained by an alternance of charging (which induces a temporary negative reward and thus seems counter-intuitive) and discharging to release the energy at a more useful time. It is also impressive to see that the granularity of action space (11) is leveraged by the controller in two cases: (1) The discharge is smooth and exactly equal to the mean consumption of the customer, with a reduction when solar becomes available;

(2) Charging is done at a small pace as soon as solar energy availability increases, then is switched to maximum at the zenith and when the energy price is at its lowest.

## 8.6. Peak Shaving and Global Adjustment

Various incentive measures have been implemented by grid operators in order to reduce peak demand. In Ontario, Canada, major electricity consumers who participate in the Industrial Conservative Initiative (ICI) are called Class A clients and are charged both an Hourly Ontario Energy Price (HOEP) as well as a Global Adjustment (GA) cost, which is usually a significant part of their annual electricity bill. The GA cost is calculated based on a customer's share of consumption during the top five peak hours of that year, as determined at *the year's end*. Class A clients are thus incentivized to reduce their electricity consumption during these five annual peak hours, referred to as "ICI hours", to significantly lower their electricity bill. This can be accomplished with an energy storage system storing energy behind the meter during non-ICI hours, then delivering stored electricity for the customer's direct use during *predicted* ICI hours.

The considered energy storage decision system is based on two components: a regression to forecast demand, and an ensemble of binary classifiers[3] trained on different subsets of the data and presenting baseline individual performances. The former predicts Ontario electricity demand 6 hours in advance, using an auto-regressive LSTM with 2 hours prediction, while the latter outputs a prediction whether the actual day presents or not the profile of a peak. The decision to use the energy storage system is then based on 2 conditions: (1) the majority of classifiers predict the day has the profile of a peak; (2) The highest forecasted demand exceeds the smallest of the current top 10 peaks (or 80% of the smallest historical peak recorded, at the beginning of the year). If both conditions return true, the system (with a delivery capacity of 6 hours) is then triggered 3 hours before the inflection point (maximum) in the prediction. On days without a peak profile, the intermittent renewable integration strategy is simply applied instead. The features used in the models are weather; time representations (both linear and cyclic) for days, weeks and months; historical data including population, holidays, irradiation and previous electrical demand from 2011 to present.

Defining our objective of capturing at least 80% of the peaks, and firing the system less than 30 times in a year, applying the decision model in a time-series fashion, i.e. training on all chronologically available data before the application year yields:

[3]Includes neural nets, KNNs, SVC, decision trees and random forests.

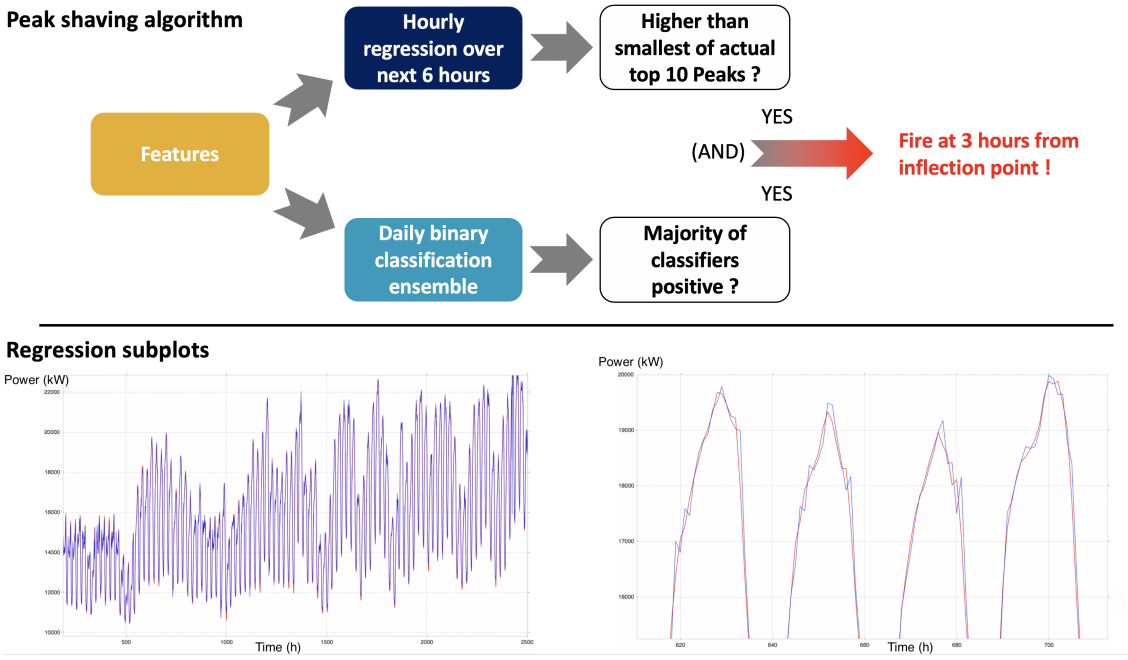


**Fig. 8.5.** Illustration of the peak shaving decision algorithm's logic (top) and the resulting regression subplots (bottom) which reached an RMSE of 347 kW. The blue line shows the LSTM's prediction, while the red line represents the real power value.

(1) 2012-2013 - 1 missed peak, 18 starts.
(2) 2013-2014 - 0 missed peak, 25 starts.
(3) 2014-2015 - 0 missed peak, 23 starts.
(4) 2016-2017 - 1 missed peak, 21 starts.
(5) 2017-2018 - 0 missed peak, 19 starts.

It is important to note that the 2016 to 2018 years were kept as reserved test sets, and were not used until the last performance assessment.

## 8.7. Electricity Market Arbitraging

The last scenario's principle is pretty straightforward, as it simply consists in buying (charging) electricity when its price is low, to resell it (discharge) at higher cost. The major additional constraint over traditional arbitraging, is the limitation induced by the energy storage's properties: capacity, energy losses, and charging and discharging efficiencies.

We train a 1200 MWh energy storage system on Ontario's HOEP data from 2014 to 2017, before applying it on the 2017-2018 year. The system's state included the current electricity price (income or losses were determined by the price at the next time iteration), the charge level of the system, the day of the year, as well as the usual time features. Figure 8.6 shows

the hourly details of this instance as well as the total virtual gains nearing the 9 million dollars.

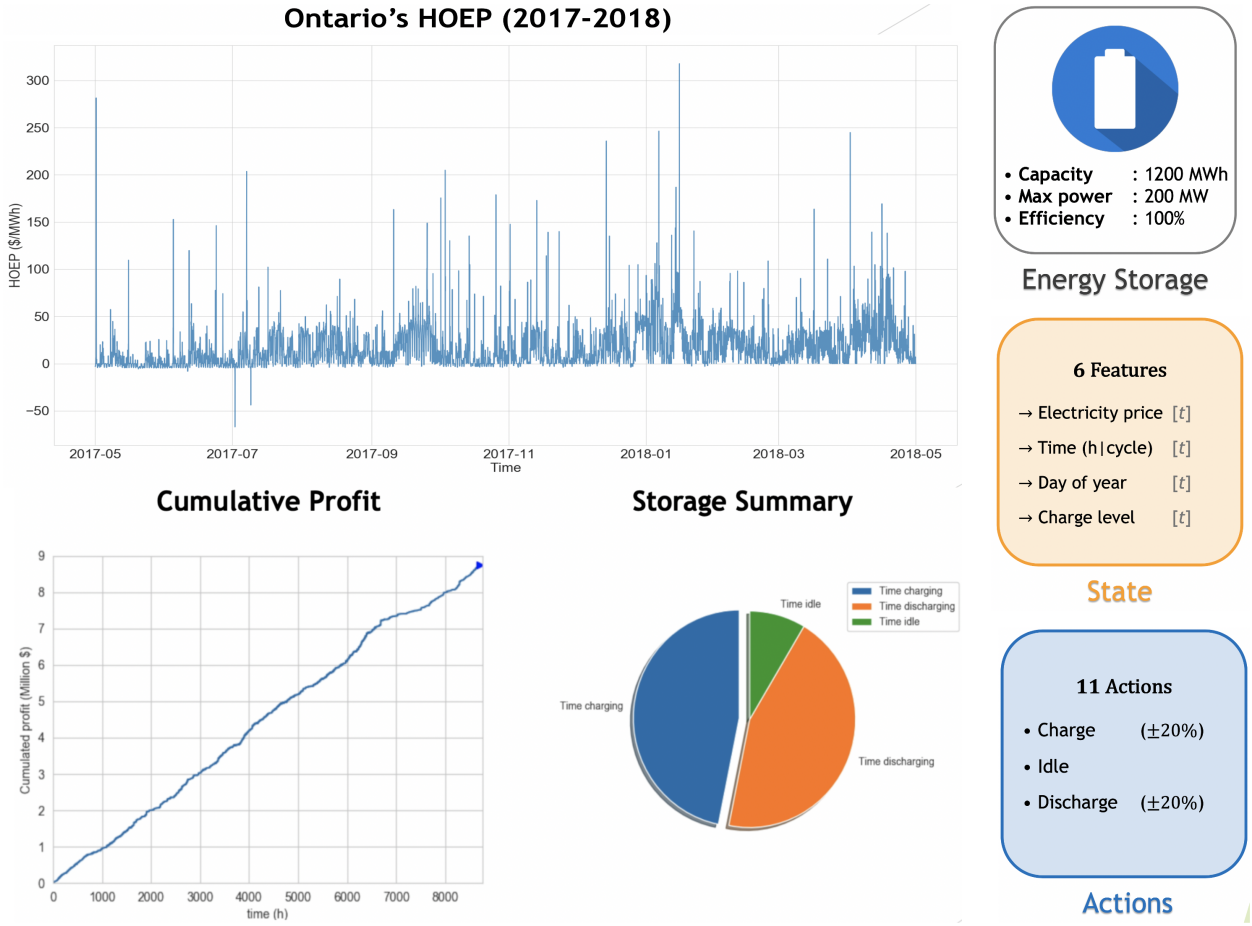


**Fig. 8.6.** Real-time simulation of an energy system, driven by an A2C controller, performing electricity market arbitraging from May 1st 2017 to April 30th 2018 on Ontario's HOEP. The top figure shows the hourly market prices over the year; the three squares on the right depicts the system's characteristics, along with its state and actions; and the bottom figures show the simulation results.

The activity breakdown pie chart illustrates the very small idle time of the system, which informally leads to the intuition that few opportunities were loss. The noteworthy point here is that the value of the horizon can be linked directly with an uncertainty and stability tolerance. Choosing a small horizon (or a smaller discount rate) results in a less risky but more stable behavior, and consequently generates a linear-looking result like the one shown above; a longer horizon (or a discount rate closer to one) on the other hand, creates more important fluctuations, but yields an overall higher gain in the long term.

# Chapter 9

# Smart Building

Energy consumed in the buildings sector, both residential and commercial, accounts for near 40% of the total worldwide energy consumption [**3**], and beyond 30% of $CO_2$ emissions [**23**]. Considering 230 billion expected square metres in new construction over the next 40 years [**22**], such numbers make smart buildings one of the major actors in the modern power grid's infrastructure, and consequently an ideal impactful candidate for the optimization and control theories application.

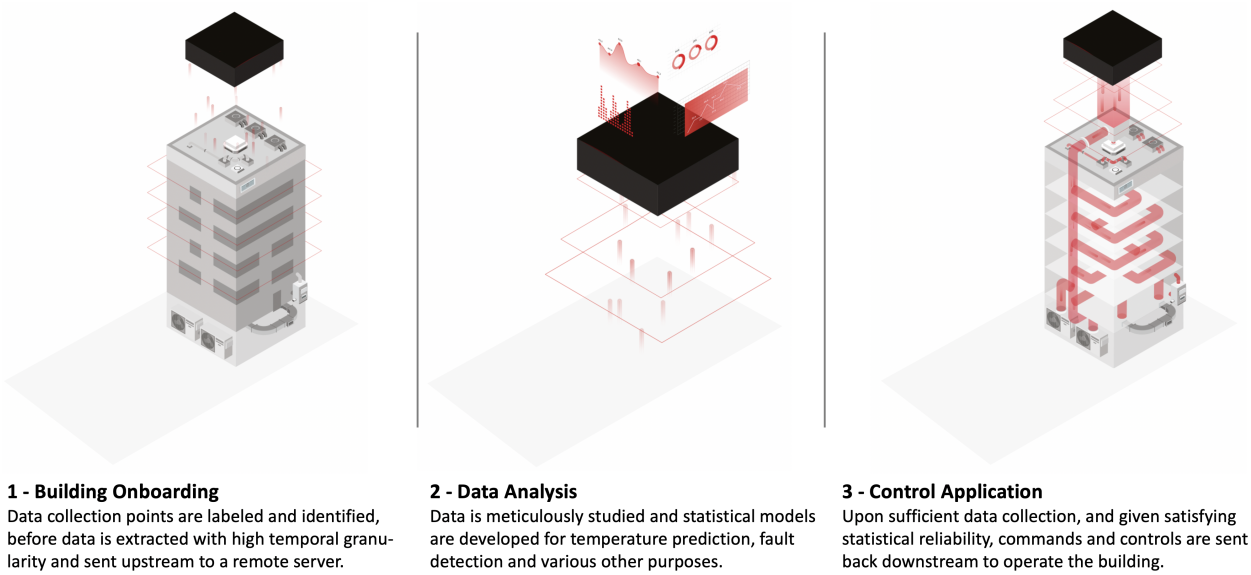


**Fig. 9.1.** Smart building optimization pipeline, from first introduction to a new building to performing remote optimal control.

From this perspective, in the present chapter we tackle the control problem of smart buildings connected to the grid, which can be seen as a complex energy storage system and thus, as an extension of the previously presented material. The conceptual end-to-end process pipeline is depicted in figure 9.1, and relies on rich 5 minutes interval data collection rates,

resulting in high granularity samples for the deployment algorithms. Despite many ongoing ventures related to this data, in the context of this work, emphasis is placed on: (1) creating an automated smart building physics model from statistical learning, (2) performing control in said building, with respect to multiple (usually competing) objectives. This ambitious project is conducted in partnership with the industrial partner *BrainBox AI* and Mitacs, led to the presented article publication, and is still under active development with new research opportunities induced from multiple successful applications and demonstrations. To illustrate the extent of the impact resulting from this work, it is interesting to note that a Canadian reader has significant statistical chances of being physically exposed to the methodology detailed herein during his upcoming week, with numerous implementations in 3 continents and several countries worldwide.

## 9.1. Thermodynamics and Heat Transfers

Temperature is one of the physical properties of a system, proportional to the average kinetic energy of its internal components (molecules and atoms). From a more practical point of view, temperature can be conceptualized as the manifestation of thermal energy, and *heat* as the transfer of such thermal energy from one system to another. By convention, heat always flows from the highest temperature towards the lowest. The three fundamental heat transfer mechanisms are:

(1) **Conduction**: Heat flow in a solid medium, or between objects with physical contact.
(2) **Convection**: Transfer of thermal energy caused by fluid motion. The *advection* definition is intricately related and refers to the macroscopic transport mechanisms of the fluid itself from one location to another (if any).
(3) **Radiation**: Transfer of energy originating from electromagnetic radiation.

In an elegant analytical perspective, these mechanisms can be combined as a sub-case of the general *transport equation* of some physical quantity $u$, which is our temperature in the present context:

$$\frac{\partial u}{\partial t} = \nabla \cdot (\alpha \nabla u) + \beta \cdot \nabla u + \gamma u + s \ , \tag{9.1.1}$$

where $\alpha$, $\beta$ and $\gamma$ represent respectively the diffusion, advection and decay coefficients, and $s$ a spatial source (or sink) term.

Focusing on the diffusive term, heat transfer in a solid medium follows the relation

$$q_k = -kA\frac{dT}{dx} \ , \tag{9.1.2}$$

referred to as *Fourier's law of conduction*, where $T$ (K) is the temperature, $A$ ($m^2$) is the area through which heat is transferred, $k$ (W/m K) the thermal conductivity[1], and the negative sign is a consequence of heat flowing from higher to lower temperatures. Considering a steady-state one-dimensional heat flow between two points of length $L$ leads to:

$$\frac{q_k}{A}\int_0^L dx = -\int_{T_{hot}}^{T_{cold}} k dT = -\int_{T_1}^{T_2} k dT \ . \tag{9.1.3}$$

In this equation, $x = 0$ is uniform at $T_{hot}$ and $x = L$ is uniform at $T_{cold}$. Furthermore, if $k$ is independent of $T$, we end up with

$$q_k = \frac{Ak}{L}(T_{hot} - T_{cold}) = \frac{\Delta T}{L/Ak} \ . \tag{9.1.4}$$

It is interesting to apply an analogy between heat flow and DC electric circuits, where the quantity $L/Ak$ can be viewed as a thermal resistance $R_k$ (K/W). In that setting, the current $i$ is equal to the voltage potential $E_1 - E_2$ divided by the electrical resistance $R_e$, just like the flow rate of heat $q_k$ is equal to the temperature potential $T_1 - T_2$ divided by $R_k$. This development is illustrated in figure 9.2.

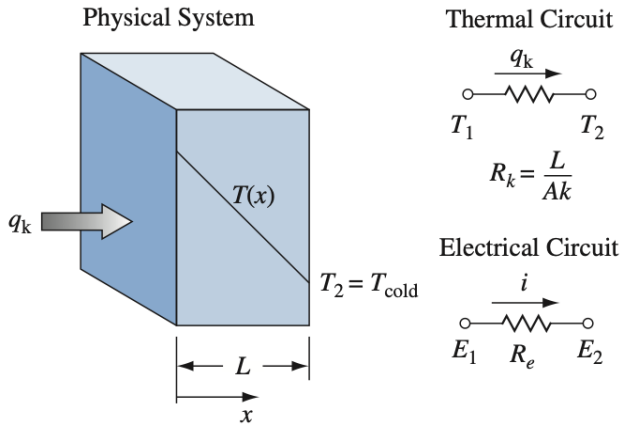


**Fig. 9.2.** Illustration of the thermal and electrical circuits analogy, and temperature distribution for steady-state conduction in a solid medium [**42**].

For the convection case, heat transfer can be divided in two simultaneous contributions: energy transfer due to molecular motion (conductive mode), and energy transfer by the macroscopic motion of fluid parcels. The latter may be caused naturally by density gradients, which we refer to as *natural convection*, and also by pressure differences induced by a pump or a fan for example. While the analytical formula is very similar to the conduction case, the physical nature of a fluid complexifies the full resolution because of the dependency on the

[1]Despite few theories on analytical derivation of this quantity, it is usually empirically measured and tabulated as the property of each material.

velocity profile $u(y)$ and turbulence, that would normally require dealing with the *Navier-Stokes equations* in a rigorous approach. For example, as depicted in figure 9.3, measuring the heat transfer between a fluid on a solid medium is given by

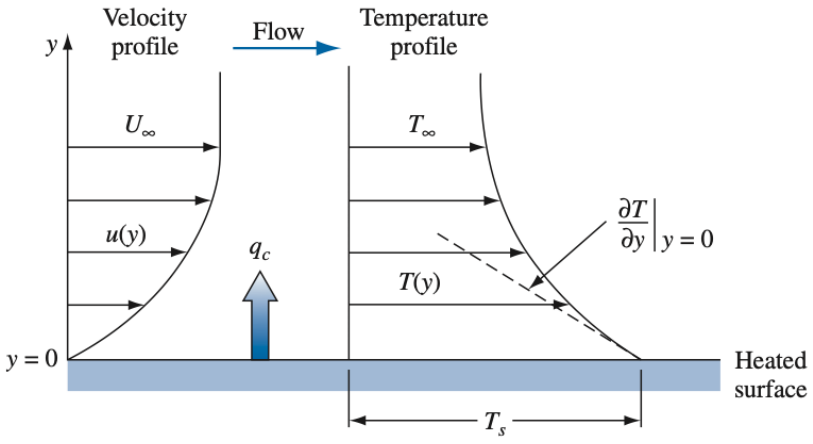


**Fig. 9.3.** Temperature and velocity profile of convective heat transfer between a surface and a moving fluid [**42**].

$$q_c = -k_{fluid} A \left| \frac{\partial T}{\partial y} \right|_{\text{at } y=0} , \tag{9.1.5}$$

but the temperature gradient depends on the motion of the fluid. To circumvent the problem, it is convenient to derive and tabulate an average convection heat transfer coefficient $\bar{h}_c$ depending on the empirical properties and conditions of both the fluid and the surface. The following equation can then be applied:

$$q_c = \bar{h}_c A \Delta T \ , \tag{9.1.6}$$

where $q_c$ (W) is the rate of heat transfer by convection and $\Delta T$ the difference between the surface temperature of the solid and the temperature of the fluid *far away* from the surface. Finally, in a similar fashion to conductance, we can derive the *convective* resistance of a thermal DC circuit as $1/\bar{h}_c A$ (K/W).

The last heat transfer mechanism, radiation, depends on the absolute temperature and nature of the surface. From the *blackbody* theory, radiant energy is emitted from a surface at a rate

$$q_r = \sigma A T^4 \tag{9.1.7}$$

where $\sigma$ is a dimensional constant equal to $5.67 \times 10^{-8}$ (W/m$^2$ K$^4$). Considering the imperfect absorption and radiation of real mediums, the previous definition normally has to be adjusted

by dimensionlesss modulus. But like in the convective case, it is usually simpler in practice to go with

$$q_r = \frac{T_1 - T_2}{\bar{h}_r A} = \frac{T_1 - T_2}{R_r} \ , \tag{9.1.8}$$

$R_r$ representing an empirical radiation resistance, which depends on the considered interacting components (we neglect its full computation and hide it in $\bar{h}_r$ constant, for simplicity purposes and because we can use empirical measurements for it).

To perform calculations on combined heat transfer systems, like the buildings application we are considering, engineering literature works with *thermal conductances* $U$ (W/K), which are the reciprocal of thermal resistances. Formally, for the three heat transfer mechanisms they are given by:

$$\text{Conduction } U_k = Ak/L \ , \tag{9.1.9}$$

$$\text{Convection } U_c = \bar{h}_c A \ , \tag{9.1.10}$$

$$\text{Radiation } U_r = \bar{h}_r A \ . \tag{9.1.11}$$

Such components are then used to model thermal circuits with capacitances, sources and inductances, which present the same mathematical properties in series and in parallel as in traditional electrical circuits.

Starting from these basic thermodynamics principles, for this application instance we consider their complex interactions inside a building or a closed structure. That is, we are interested in quantifying as accurately as possible the internal temperatures of different spatial points, which vary continuously over time, as they are affected by several factors such as air flow, heating, ventilation and air conditioning (HVAC) controls, human bodies and activities, sun reflections and projections, and isolation to cite only but a few. While decades of engineering research and applied physics modeling provided relatively accurate tools and models, deploying and scaling such methods is particularly expensive both in terms of financial and human resources, and unrealistic for large-scale applications.

## 9.2. Temperature Prediction

Modern buildings and their electronic monitoring systems count several thousands, if not millions, of frequently-sampled different measurement types. As introduced in chapter 7, such large-scale reality combined with multi-disciplinary teams implication in the data acquisition process, requires the definition of rigorous yet universal methodologies and labeling systems. While the HVAC community already relies on different tagging

standards, with the most adopted one being from the official American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE) [**5**], we successfully deploy in parallel a simpler OR-based tagging system for statistical learning and control convenience. More specifically, upon *onbarding* a new building, engineers (or algorithms) throughout the world performing the installation of data acquisition must tag each measurement type according to the dynamic system components introduced in section 1.3:

- **States (X)**: quantities of interest describing the properties and internal characteristics of the building itself. Represent mainly internal temperature measurement points.
- **Controls (U)**: all *manageable* entities, directly *influencing* the state variables. These include HVAC components, which are then subdivided whether they are continuous or discretized. All their operating properties related to power, energy and other environment-impactful resources are documented and linked for optimization, control and tracking purposes.
- **Disturbances (W)**: *uncontrollable* parameters also affecting the state variables, such as occupancy measures, humidity, external weather, time features etc.
- **Others**: variables (usually specific to the considered building) that may be useful but requires human analysis for proper classification.

While far from perfect, such definition is broad enough to be applied in practice by any individual, regardless of its professional competencies or academic specialization.

Once properly identified, and with enough historical data (usually around 4-6 weeks), all samples stored in the data base are manipulated in groups according to their respective categories or subdivision. Usual data treatments and analysis are applied: outlier detection, interpolation for small data gaps and data subsets for larger ones, one-hot encoding for categories and normalization to enumerate only but a few. These "clean" points are then processed automatically into a supervised learning format for temperature prediction, according to the following steps:

(1) *State* variables are placed as regression targets, duplicated, concatenated and shifted along the time dimension to match the desired *prediction horizon* and granularity (usually around 2 hours with 5 minutes intervals).
(2) *State* variables are also used as features, but only up to the current time step, since we predict their value at time index $t + 1$ and onwards.

(3) *Control* variables are duplicated and shifted to augment the features matrix with both their historical and future values (to simulate different trajectories resulting from different controls).
(4) *Disturbances* are also included in the features matrix, with the columns known in the future time steps (like for weather prediction or time features) treated identically as for the controls.
(5) Resulting matrices are then reshaped into (lists of) tensors to match the (multi-headed and multi-tailed) deep learning architecture(s) chosen for the prediction task, with a specific care to include all available knowledge in the input features.

An example of the aforementioned automated conversion process from historical time series data is illustrated in the figure below.

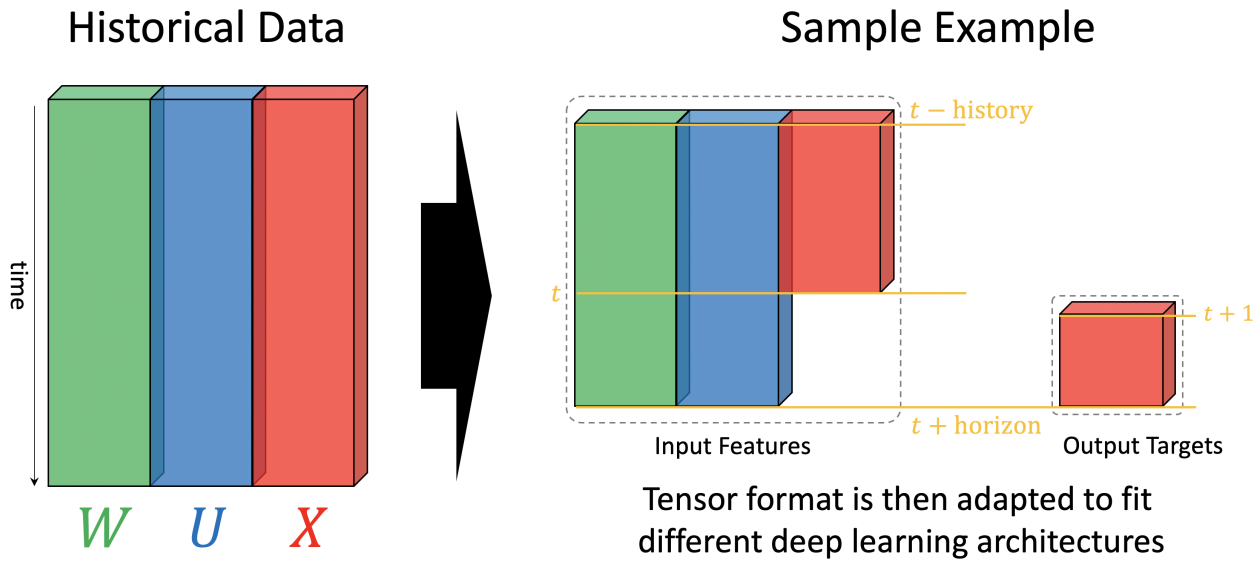


**Fig. 9.4.** Automated supervised learning conversion example from historical time series data, using dynamic system components X (states), U (controls) and W (disturbances) to perform functional modeling.

Using multiple channels for the inputs, in a deep multi-modal learning fashion, allows an efficient combination of all data types regardless of their nature or temporal granularity[2]. Furthermore, this separation allows channel-specific operations like convolutions, which can later be re-used as feature detectors or for transfer learning purposes. This approach can also be used on the targets, creating separate output channels for each individual zone or floor of the considered building, just like in the multi-tailed DQN presented in section 7.2.1.

[2]Some input variables, like weather predictions, have hourly granularity unlike the vast majority of the remaining data which has 5 minutes sampling frequency.

A single model is usually sufficient to predict all temperature points in small or medium-sized buildings, but larger instances like skyscrapers or commercial centers require division into sub-models, based on the physical separations of the building (in terms of floor, zones or rooms, and specific HVAC equipment reach out). With this in mind, starting from a huge single model for all zones, we use a recursive building hierarchy to break the prediction into progressively smaller and smaller submodels, until a satisfying validation error is obtained for all zones. Naturally, individual problematic or biased zones can be ignored or treated with error fitting depending on the circumstances. Applying all the described development, in conjunction with the methodologies and deep learning architectures detailed in chapter 7, allows us to achieve predictions with an RMSE typically in the range of 0.2°C for 2 hours ahead predictions[3]; which is lower than the usual temperature sensor uncertainty of ± 0.3°C. The best resulting deep learning architectures are: (1) the multi-scale convolutional recurrent neural network with a slicing window (MCNN-RNN) for its efficiency (good performance vs low calculation time), and deep bidirectional GRUs with skip connections and Attention in terms of pure RMSE minimization. Different model types are deployed depending on the calculation burden and prediction precision requirements for each building.

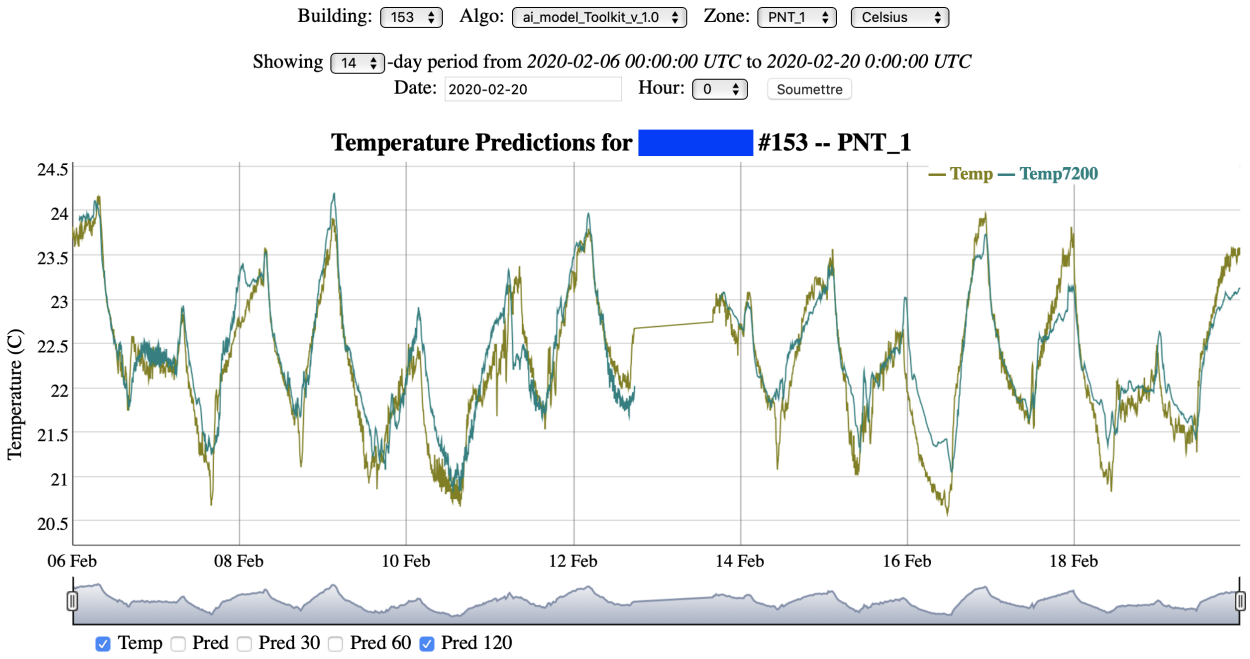


**Fig. 9.5.** Internal temperature observed (brown-green) and 2 hours prediction (blue-green) for a room in an anonymized medium-sized building in Montreal (QC), Canada. Connection with the building was temporarily lost on February 13th, which explains the absence of prediction on this day.

[3]Prediction accuracy also fluctuates with the seasonality, as the distribution changes.

Pushing the analysis further, we realize that despite relatively low classical test error in deployment, the forecasting model lacks sensitivity to controls variation: triggering full heating does not result in the expected physical response, i.e. temperature increase, and vice versa for cooling. The conclusion is that the learning process generalized poorly, and relies too heavily on historical values such as the present temperature and time inputs to guide future predictions. Back to the drawing board, we inject some more physics into the methodology, with the two following modifications:

(1) Replacing the absolute temperature prediction $T$ by a regression on the temperature *difference* $\Delta T$ between each sampling interval. While this approach is common and proven practice in several machine learning literature, in the present context it also has an important physical meaning, as heat flow is based on temperature differences. Considering this, we also add different temperature gradients to the input features (difference between inside and outside air temperature, between zones, temporal variations ...).
(2) Adding *enthalpy* as a feature. In its most fundamental form, enthalpy is defined as $H = U + pV$, where $U$ is the total energy of a thermodynamic system, and $pV$ (pressure of the system times its volume) the additional work required to displace the atmosphere for its creation. But one can show with the proper thermodynamic equations manipulations, which we omit for the sake of concision, that this quantity is directly linked to the heat balance of the system. In practice, we use the CoolProp package [**7**] to compute the fluid properties of air at a given temperature, humidity, and at sea level pressure of 101.3 kPa.

Not only do these two additions directly solve controls sensitivity, they also reduce prediction error down to an average RMSE of 0.15°C. It is particularly worth noting that the inflection points in figure 9.6 are always consistent with the real ones for the $\Delta$T model, which is very interesting for autoregressive applications and RL, or model-based predictive control (MPC). Lastly, expressing the building's state variables this way is a lot more data-efficient and solves the seasonality problem, as temperature difference logic holds regardless of the outside temperature, and thus enables year-round deployment on weather never seen prior in training.

Continual real-time connectivity and tracking of thousands of points is a major technological advance, but evidently suffers from sporadic communication interruptions or signal losses. While complete communication disconnection compromises the entire pipeline, sporadic or expected missing entries in deployment can be elegantly solved. In the former case, as the information sweep is performed every 5 minutes, if one of the entries is missing or if

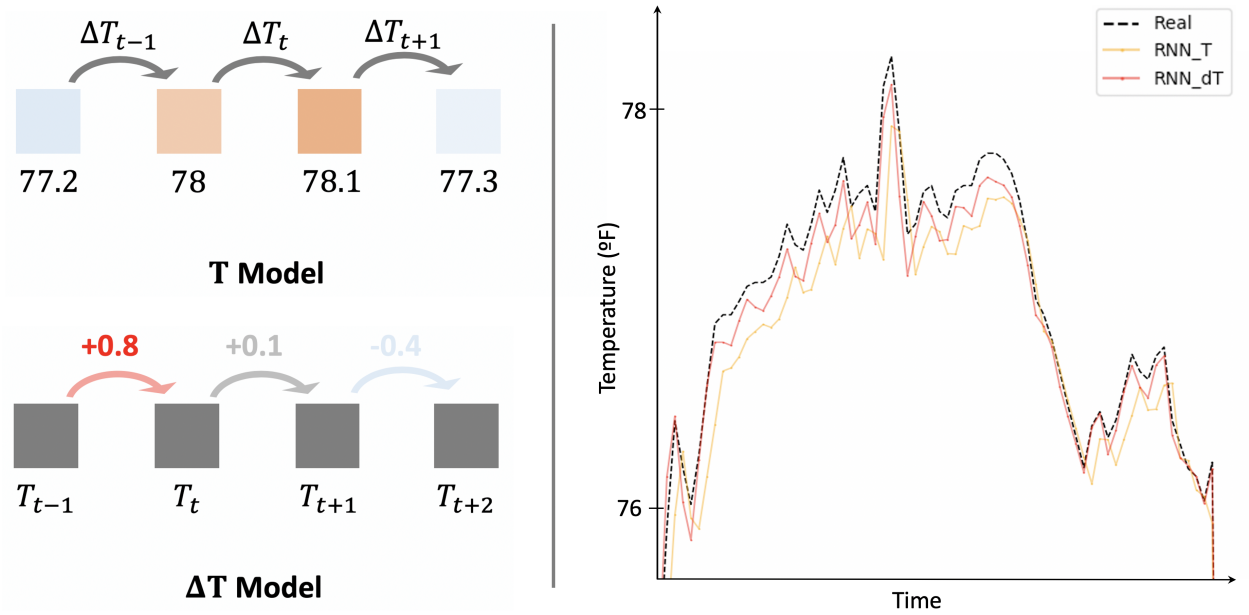


**Fig. 9.6.** Temperature (T) and temperature difference ($\Delta$T) models principle (left) and application in a building (right). The black dashed line represents real temperature in a zone sampled every 5 minutes, while the yellow and red lines show 1 hour predictions using both the T and $\Delta$T models with a single-layer LSTM.

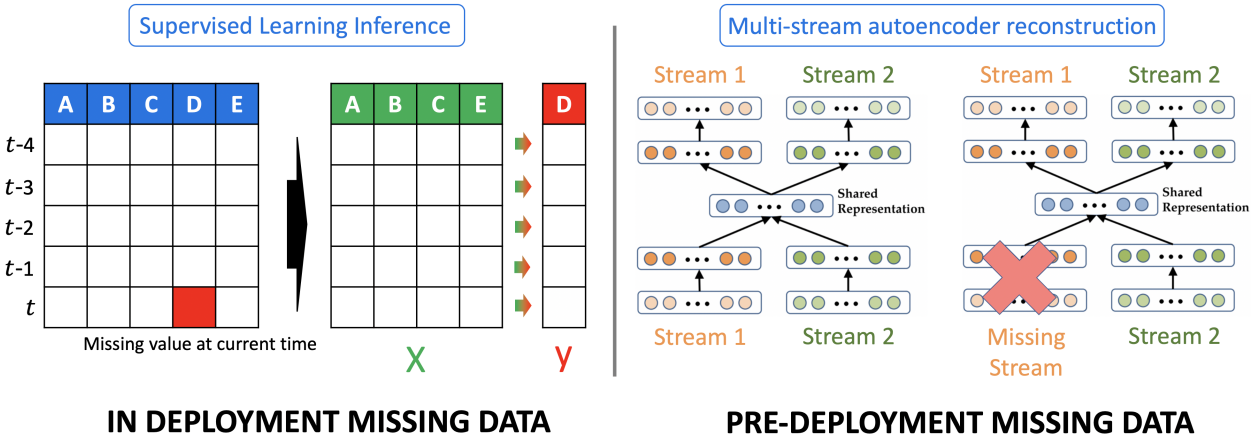


**Fig. 9.7.** Missing data treatment for deployment of prediction, whether the interruption arises *during* deployment, or is *expected prior* to deployment (right).

a fault is detected: (1) signal is sent to competent authorities to solve the problem; (2) the missing column is immediately taken and used as target for a supervised learning instance with a simple MLP, placing other columns as features. Triggering this gets the prediction back in line in less than 15 minutes, and allows control algorithms to continue they work seamlessly. In the other case, when a specific data type is *expected* or known to be occasionally missing prior to training, multi-stream autoencoder reconstruction pre-training is performed as detailed in figure 9.7. While only normal auto-encoders have been explored up

to now, a lot of improvement is possible in this area with modern literature around variational and denoising autoencoders. Finally, while these two solutions do not have rigorous guarantees, they do perform well in practice.

## 9.3. Autonomous Control

In this section we consider the multi-objective problem of optimizing expenses, thermal comfort, power peaks, energy consumption and equipment cycling in smart buildings equipped with multiple air-handling units (AHU) such as Roof Top Units (RTU) and connected to the electrical grid. We assume control will be applied with high granularity (decision epochs every 15 minutes), from a remote low-computational power device having local access to all the heating, ventilation and air-conditioning (HVAC) components inside the building (see figure 9.8). For application realism purposes, we also consider sparse intermittent connectivity with the remote control device, making it independent from the central computing source except for occasional data transfers. Finally, we impose a set of pre-defined fixed constraints on thermal values for the safety of the users inside the buildings, which will automatically trigger a fall-back position to classical controls if needed.

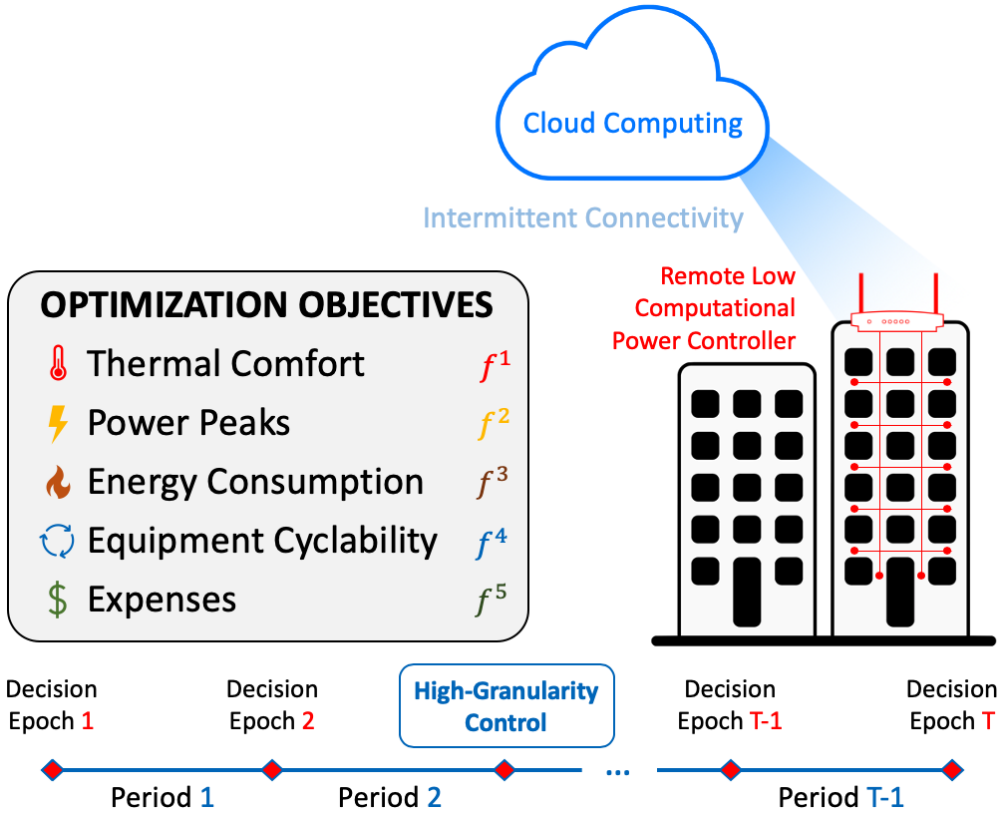


**Fig. 9.8.** Remote high-granularity control problem considered.

The considered multi-objective optimization problem can be expressed as

$$\max \sum_{t=1}^{T} \left\{ \alpha_t^1 f_t^1 + \alpha_t^2 f_t^2 + \alpha_t^3 f_t^3 + \alpha_t^4 f_t^4 + \alpha_t^5 f_t^5 \right\} \ , \tag{9.3.1}$$

where $t$ indexes time up to a horizon $T$, and $f_t^i$ and $\alpha_t^i$, $i \in \{1,2,...,5\}$, are the different objective functions as depicted in Fig. 9.8, and their respective weighting coefficients. This optimization is subject to the thermodynamics of the building, the occupants' safety measures, and the technical specificities of the HVAC equipments.

### 9.3.1. Thermodynamics Modeling

Starting from the transport equation 9.1.1, we start by focusing on the diffusion term

$$\frac{\partial u}{\partial t} = \nabla(D \cdot \nabla u) \ , \tag{9.3.2}$$

where $D = \alpha$ is the *diffusion coefficient* ($\mathrm{m^2 s^{-1}}$). Such equation is an example of the general *conservation equations* of the form:

$$\frac{\partial u}{\partial t} + \nabla(F) = 0 \ , \tag{9.3.3}$$

with $F$ denoting what we call the *diffusive flux*. Such equation is very broad and describes a large range of physical phenomenons including thermal conduction, passive chemical substance dispersion, magnetic field penetration in conducting substances and a lot more. With a constant $D$ and in a single dimension, the equilibrium solution can easily be computed by $u_{eq}(x) = ax + b$, where the value of the integration constants $a$ and $b$ are fixed by the boundary conditions of the problem.

Defining a spatial and temporal meshing as follow[4]:

$$x_j = j \times \Delta x \ , \quad j = 0,1, ..., J-1 \ , \tag{9.3.4}$$

$$t^n = n \times \Delta t \ , \quad n = 0,1,..., N \ , \tag{9.3.5}$$

and given the initial and boundary conditions, the most intuitive approach consists in converting the main equation into a finite difference for the temporal derivate, and a centered temporal difference of second order for the second derivative of $x$. This leads to the very well known *Forward-in-Time-Centered-in-Space* (FTCS) algorithm, which an be seen as a equivalent of the Euler method for partial differential equations (PDEs):

[4]The time iteration index $n$ goes up to $N$ because $n = 0$ is used for the initial condition.

$$\frac{u^{n+1}-u^n}{\Delta t} = D\frac{u_{j+1}-2u_j+u_{j-1}}{(\Delta x)^2} \quad \{\text{Finite Difference}\} \ , \tag{9.3.6}$$

$$u_j^{n+1} = u_j^n + \left(\frac{D\Delta t}{(\Delta x)^2}\right)(u_{j+1}^n - 2u_j^n + u_{j-1}^n) \quad \{\text{FTCS}\} \ . \tag{9.3.7}$$

Evaluating the right term of the FTCS method at time $n+1$ instead of $n$ results in the so-called *Euler implicit* method

$$u_j^{n+1} = u_j^n + \left(\frac{D\Delta t}{(\Delta x)^2}\right)(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) \quad \{\text{Euler Implicit}\} \ , \tag{9.3.8}$$

that can be expressed under the matrix form

$$K_{ij}\ u_j^{n+1} = u_i^n \ . \tag{9.3.9}$$

In the present context, with the pure discretized diffusion equation, the matrix $K$ corresponds to the tridiagonal matrix:

$$K_{ij} = \begin{bmatrix} b_0 & c_0 & & & & 0 \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{J-2} & b_{J-2} & c_{J-2} \\ 0 & & & & a_{J-1} & b_{J-1} \end{bmatrix} \tag{9.3.10}$$

with

$$a_i = -D\Delta t/(\Delta x)^2 \ , \quad i = 1, ..., J-1 \ , \tag{9.3.11}$$

$$b_i = 1 + 2D\Delta t/(\Delta x)^2 \ , \quad i = 0, ..., J-1 \ , \tag{9.3.12}$$

$$c_i = -D\Delta t/(\Delta x)^2 \ , \quad i = 0, ..., J-2 \ . \tag{9.3.13}$$

One should pay careful attention upon incorporating the problem-specific boundary conditions, as it affects the definition of both the matrix and $u_i^n$.

Von Neumann numerical stability analysis consists in observing under which condition a small perturbation of the form

$$u_j^n = \xi^n \exp(ikj\Delta x) \ , \tag{9.3.14}$$

is amplified or dampened by a given numerical algorithm. It is important to understand the meaning of the value above: the product $j\Delta x$ is the discretized version of our spatial variable

$x$, so $\exp(ikj\Delta x)$ corresponds to the harmonic mode of wavelength $\propto k^{-1}$. The (complex) amplitude of this mode is given by the $\xi^n$ term, and will grow with time - as discretized by $n$ - if $|\xi|^2 \equiv \xi * \xi > 1$, where $*$ denotes the complex conjugate. The numerical stability of an algorithm like the one we introduced can thus be tested by verifying the condition $|\xi| < 1$ for all values of $k$ in the spatial domain. Our choice of the Euler implicit method relies on the powerful result that

$$\xi = \left(1 + 4\frac{D\Delta t}{(\Delta x)^2}\sin^2(k\Delta x/2)\right)^{-1} \ , \tag{9.3.15}$$

which implies that $|\xi| < 1$ for any combination of $\Delta t$ and $\Delta x$. This result can be explained by the coupling of all the nodes in the calculation. It is however important to remember, from a physical perspective, than even if numerical stability is assured a time step and a spatial step sufficiently small are required for an appropriate precision to capture the underlying physics phenomenon at stake. Lastly, the Euler implicit method is the one suggested by literature for applications with bigger disparities between characteristic times, which is typically the case in the building HVAC scenario.

Turning our attention on the specific implicit finite-elements heat transfer implementation, we consider that each given node $i$: is exchanging heat with all neighbouring nodes $j$ and $k$ (the latter indexes nodes with known defined temperature, a boundary condition) through conduction, convection and radiation, expressed as an equivalent conductance $U$; has *capacitance* or *thermal mass* $C = mc_p$ (J/K) [5], where $c_p$ is the *specific heat*[6] of the medium; is receiving heat from a source $Q$. The finite difference equation can then be written:

$$\sum_j U_{ij}^{t+1}(T_j^{t+1} - T_i^{t+1}) + \sum_k U_{ik}^{t+1}(T_k^{t+1} - T_i^{t+1}) - \frac{C_i}{\Delta T}(T_i^{t+1} - T_i^t) + \dot{Q}_i^{t+1} = 0 \ , \tag{9.3.16}$$

where $U_ij$ (W/K) is the conductance between nodes $i$ and $j$ (see equation 9.1.9 for specific formulas in the conduction, convection and radiation settings), $\dot{Q}$(W) the heat flow into the node and $\Delta t$ the time step, in seconds. Defining $N$ as the number of internal nodes, and $M$ as the number of nodes with known boundary temperatures, the matrix form is implemented using the upper triangular matrix $U$ (due to symmetry), a vector $C$, known connection to temperature sources as a matrix $S$ and heat flow into the nodes in the vector $Q_{in}$.

### 9.3.2. Case Studies

We perform our proof of concept through building simulations by accounting the interaction of all the major thermodynamics actors in the system: shared walls and doors

[5]The capacitance of an object is a measure of how much heat it can store.
[6]Specific heat is the amount of heat a medium requires to have a one degree temperature differential.

between zones, open spaces and room content, external temperature, users activities, solar radiance, and HVAC components behavior. Real historical weather temperature is used to drive the boundary conditions of the simulation, internal thermal properties are chosen to represent archetype building types and characteristics, and the HVAC systems considered are RTUs with two heating stages and two cooling stages for a total of five possible control indexes if we include the "off" position. Finally, human activities and solar contributions are sampled from a normal stochastic distributions with parameters varying depending on each building's nature and schedule.

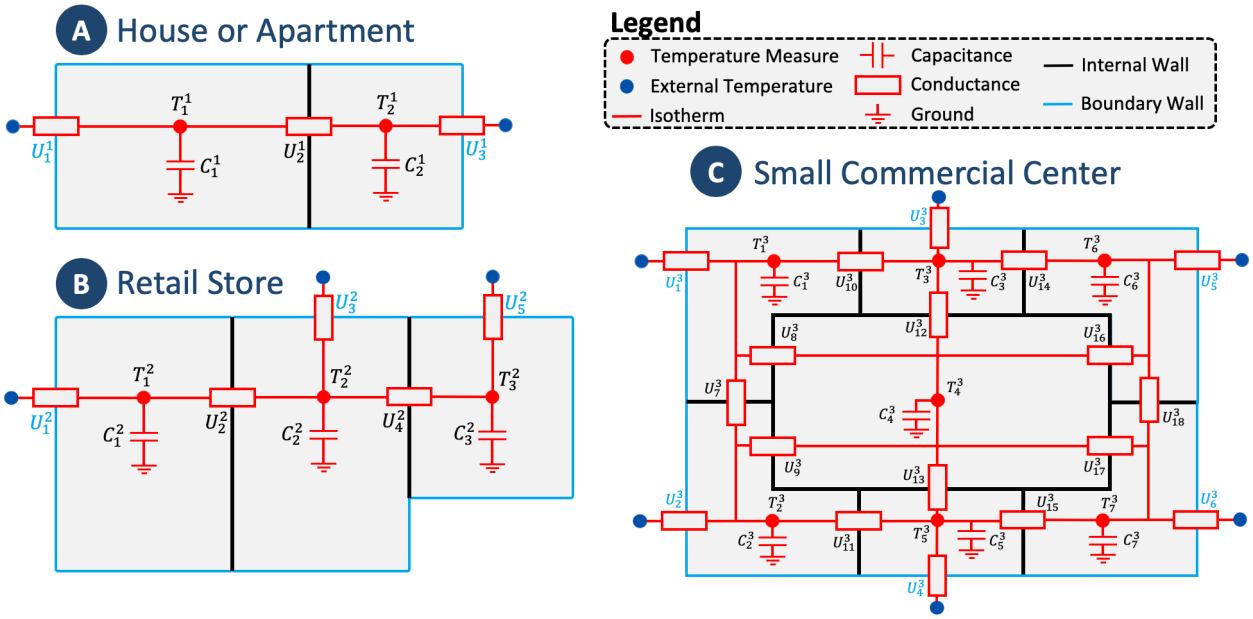


**Fig. 9.9.** Thermal RC circuit instances considered in the building simulation. Independent solar, human and HVAC heat contributions are added at every internal temperature measurement point, and the external temperature is provided from real historical data.

For building types, we consider the three following distinct instances: a house (or an apartment) with two AHUs consuming 10kW for stage 1, and 15kW for stage 2; a retail store with three similar but larger systems having a 20kW stage 2; and finally a small commercial center with seven independent controllers identical to the retail store. We assume a 90% efficiency on all equipment, meaning that 90% of the input power is converted into heating or cooling. Each building instance (see figure 9.9) is represented by its own thermal circuit and includes heat contributions from users, solar radiance and HVAC components in each individual zone (see figure 9.10 for simulation parameters). The comfort zone is defined to be between 19.5°C and 22.5°C, but while it is kept constant in instance A, set points are scheduled in the two other instances to adjust the dead band between 16°C and 26°C during the non-occupancy hours, from 6 PM to 6 AM. The unitary kWh electricity price

varies during the day, taking the value of 0.132$ from 7AM to 11AM, 0.095$ from 11AM to 3PM, 0.132$ from 3PM to 7 PM, and 0.05$ in the remaining hours. Power is charged based on the month's highest peak, at the rate of 14.58$/kW. While a realistic simulation is targeted, our main objective is also to introduce a high level of variation and uncertainty, to illustrate the generalization and adaptation capacity of our DRL controller.

| | |
|---|---|
| **INSTANCE A** | |
| $U_1 = 300, U_2 = 100, U_3 = 250$ | W/K |
| $C_1 = 11 \times 10^6, C_2 = 11 \times 10^6$ | J/K |
| **INSTANCE B** | |
| $U_1 = 400, U_2 = 1000, U_3 = 200, U_4 = 750, U_5 = 300$ | W/K |
| $C_1 = C_2 = 12 \times 10^6, C_3 = 10 \times 10^6$ | J/K |
| **INSTANCE C** | |
| $U_1 = U_2 = U_3 = U_4 = U_5 = U_6 = 200$ | |
| $U_7 = U_{10} = U_{11} = U_{14} = U_{15} = U_{18} = 500$ | W/K |
| $U_8 = U_9 = U_{12} = U_{13} = U_{16} = U_{17} = 1000$ | |
| $C_1 = C_2 = C_6 = C_7 = 11 \times 10^6, C_4 = 12 \times 10^6$ | |
| $C_3 = C_5 = 10 \times 10^6$ | J/K |

**Fig. 9.10.** Value of the parameters used in the simulation.

To increase application realism, we perform training in a centralized fashion from a cloud computing resource, which then transfers the weights matrix to the local remote controller. During deployment, only the results of the forward pass are accessible to this device, with weight updates possible only through a pre-defined schedule. To assess performance on different deployment steps, with increasing amounts of available data, we divide the simulation into two phases:

- Phase 1: One continuous month deployment with an initial training on two months of similar conditions.
- Phase 2: Year-round deployment, with 8 months of initial training data and no weight update.

Phase 1 is evaluated on January 2020 (one of the coldest months) and trained from November 2019 to the end of December 2019, while phase 2 is trained from July 7, 2018 to March 14, 2019, then tested from March 15, 2019 to March 15, 2020 (see figure 9.11).

For the RL side of the simulation, the environment's partial observation includes the following information from the previous hour up to 15 minutes before the present time: temporal iteration's value $t$, and cyclic features of the form $\sin(2\pi t/T)$ and $\cos(2\pi t/T)$ where $T$ is the episode's length; the HVAC systems' index; internal and external temperature;

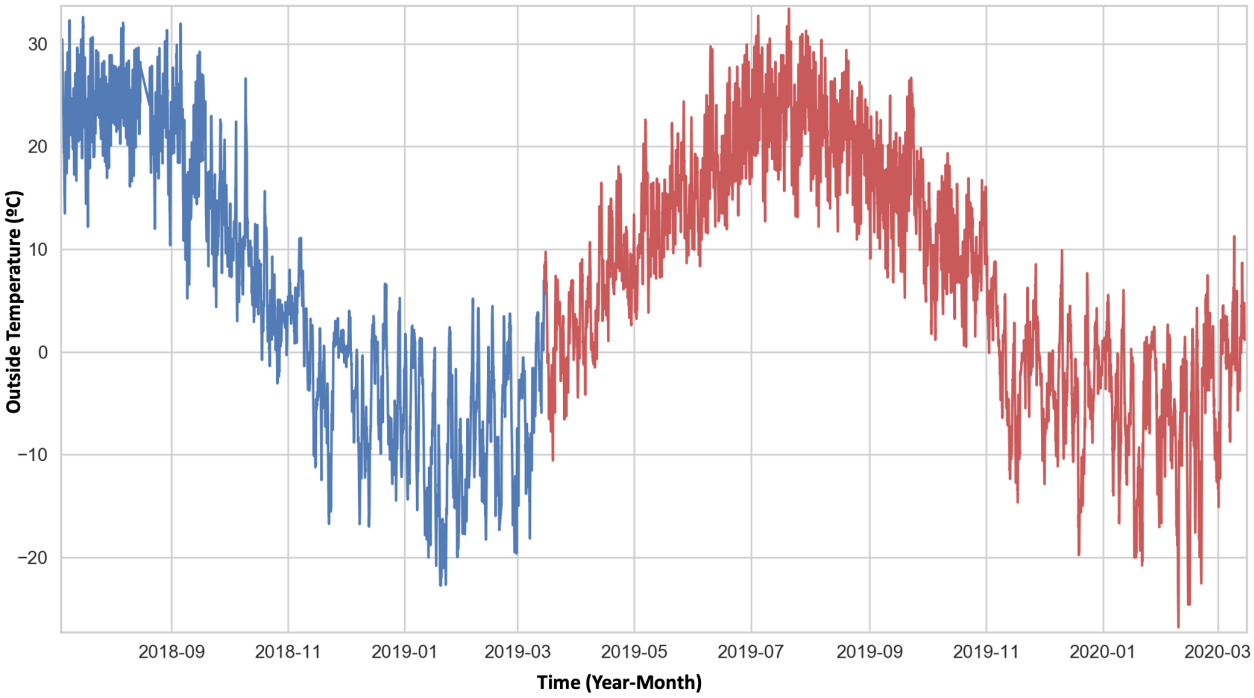


**Fig. 9.11.** Outside temperature in Montreal (QC), Canada, from July 7 2018 to March 15 2020. Data originates from the Dark Sky API [**1**], and the training data for phase 2 is depicted in blue, while the test deployment data is shown in red.

temperature set points; energy consumed in each zone; power calls in each zone; and highest building power peak since the beginning of the month. Three actions are considered for each HVAC controller: heat (increases control index), do nothing (leaves index unchanged), or cool (lowers control index). Cooling stage 2 is represented by the lowest control index 0, while heating stage 2 is indexed by the maximum, 4. The reward process is built as follows: the agent receives a periodic reward of 50 at each time step while being between the temperature set points, and -20 if outside of the range, to stimulate comfort; -15 reward for each action different from “do nothing", to reduce useless toggling; a negative reward proportional to the power consumption each time the current highest power peak of the month is exceeded; a penalty linearly scaling to the energy cost at each time step; and finally a -1000 penalty each time the temperature is 2 degrees above or below the set points. As a safety measure, the agent is then replaced by classical controls until the temperature is back within comfort range. The resulting mathematical system for the agent is to maximize its return for a 24-hours period, while being exposed to all the combined reward mechanisms described above.

The training phase of the modified DQN consists in 50 000 epochs of 24-hours episodes simulation, using an $\epsilon$-greedy policy where either a random action is chosen for every controller with probability $\epsilon$, or a stochastic policy using the vector of normalized $Q$-values as

a probability distribution over the actions (sometimes referred to as *Boltzmann Policy* [**28**]) is applied instead. Using this methodology bolsters appropriate exploratory behavior in the long term, while still allowing convergence towards a final optimal random policy if need be (unlike the original algorithm always deterministically using the *arg max* of the Q-factors). Starting with $\epsilon = 1$, we perform a linear decrement to reach a floor value of 0.05 at 80% of the training epochs. Optimization is achieved with the *Adam* [**38**] gradient descent algorithm, with constraints to limit the gradient norm between -1 and 1 to avoid instability arising from a major update. Finally, a scheduled learning rate is applied with a starting value of 0.001, reduced to $5 \times 10^{-4}$ at 30% of training, then finally set to $1 \times 10^{-4}$ from 60% to the end of the computation.

### 9.3.3. Results and Discussion

We assess the test simulations by defining Key Performance Indicators (KPI) with respect to each initial target objective: total and individual costs for the expenses; average daily discomfort time for thermal comfort; average daily energy consumed for energy consumption; highest overall peak for power peaks; and total number of performed cycles for equipment cyclability. Following these definitions lead to the conclusion that the multi-objective optimization is successful overall, as the DQN controller outperforms or equals its classical reactive peer for all KPIs in phase 1 (see figure 9.12), and nearly all in phase 2 (see figure 9.13).

The only poorer performances in the year-round deployment are the ones related to discomfort and energy. It is however important to note that the difference between both controllers is smaller than the model's time step in the first case, making it negligible in the context of this simulation, while in the second case energy cost savings were still observed despite slightly more consumed energy. Running more training epochs or having a complete year of data would probably solve or improve these aspects. Lastly, it is worth mentioning from a safety perspective that security measures were not triggered at any point for phase 1, and only twice for phase 2 during the coldest days of winter, illustrating the ability to operate reliably within strictly established constraints.

Figure 9.14 visually depicts a combination of important optimal HVAC behaviors autonomously learned by the DRL controller, which match suggested theoretical guidelines found in the HVAC literature [**5**], [**83**]: (1) temperature barely touches the set points and reacts pre-emptively just before doing so, (2) pre-heating in the morning is linear and with a precise phase shift, yet respects the more restrictive set point right on time during the occupancy schedule, (3) the central zone never triggers HVAC as it knows it will benefit

| Measure | Instance A | | Instance B | | Instance C | |
|---|---|---|---|---|---|---|
| Total Cost ($) | 841.8 | 865.7 | 1303.7 | 1331.6 | 1240.4 | 2266.2 |
| Power Cost ($) | 437.4 | 437.4 | 874.8 | 874.8 | 1020.6 | 2041.2 |
| Energy Cost ($) | 404.4 | 428.3 | 428.9 | 456.8 | 219.8 | 225.0 |
| Average Daily Discomfort Time (Minutes) | 31.5 | 106.25 | 43.2 | 108.7 | 39.2 | 91.1 |
| Average Energy Consumed Daily (kWh) | 158.3 | 167.2 | 168.9 | 176.9 | 87.4 | 89.0 |
| Highest Power Peak (kW) | 30.0 | 30.0 | 60.0 | 60.0 | 70.0 | 140.0 |
| # Equipment Cycles (Cycles) | 260 | 315 | 394 | 418 | 925 | 1038 |
| : DQN : Classical | | | | | | |

**Fig. 9.12.** KPI for phase 1 deployment, comparing both the DRL controller and its classical reactive counterpart. Results are highlighted in green when DRL performances exceeds classical controls, and in red in the opposite case.

| Measure | | Instance A | | Instance B | | Instance C | |
|---|---|---|---|---|---|---|---|
| Total Overall Cost ($) | Spring | 1301.1 | 1770.3 | 2050.8 | 2952.2 | 3246.9 | 6327.2 |
| | Summer | 951.6 | 1424.0 | 1414.5 | 2755.3 | 2682.5 | 4179.2 |
| | Autumn | 1713.7 | 1972.7 | 2690.8 | 3304.5 | 3356.2 | 6436.4 |
| | Winter | 2445.0 | 2510.3 | 3527.1 | 3897.2 | 4535.5 | 6749.0 |
| Total Power Cost ($) | | 874.8 | 1312.1 | 1603.8 | 2478.6 | 3061.8 | 6123.6 |
| | | 874.8 | 1312.2 | 1312.2 | 2624.4 | 2624.4 | 4082.4 |
| | | 1093.5 | 1312.2 | 2041.2 | 2624.4 | 3061.8 | 6123.6 |
| | | 1312.2 | 1312.2 | 2332.8 | 2624.4 | 3936.6 | 6123.6 |
| Total Energy Cost ($) | | 426.2 | 458.1 | 447.0 | 473.6 | 185.2 | 203.6 |
| | | 76.8 | 111.8 | 102.3 | 130.9 | 58.1 | 96.9 |
| | | 620.2 | 660.5 | 649.6 | 680.1 | 294.4 | 312.8 |
| | | 1132.8 | 1198.1 | 1194.3 | 1273.1 | 598.9 | 625.5 |
| Average Daily Discomfort Time (Minutes) | | 44.7 | 69.8 | 44.3 | 59.0 | 53.2 | 39.9 |
| | | 27.7 | 23.3 | 13.7 | 23.6 | 28.3 | 19.5 |
| | | 33.5 | 52.7 | 54.6 | 71.67 | 53.0 | 39.7 |
| | | 40.5 | 111.2 | 61.0 | 112.3 | 49.3 | 87.5 |
| Average Energy Consumed Daily (kWh) | | 54.2 | 56.5 | 57.8 | 56.9 | 25.1 | 24.7 |
| | | 9.2 | 11.9 | 11.8 | 13.9 | 6.5 | 10.0 |
| | | 149.8 | 156.9 | 85.0 | 85.0 | 39.7 | 40.0 |
| | | 150.1 | 159.8 | 159.9 | 168.0 | 80.6 | 83.8 |
| Highest Power Peak of the Season (kW) | | 20.0 | 30.0 | 40.0 | 60.0 | 70.0 | 140.0 |
| | | 20.0 | 30.0 | 30.0 | 60.0 | 60.0 | 140.0 |
| | | 30.0 | 30.0 | 40.0 | 60.0 | 70.0 | 140.0 |
| | | 30.0 | 30.0 | 60.0 | 60.0 | 100.0 | 140.0 |
| Total Equipment Cycles (Cycles) | | 704 | 729 | 879 | 918 | 1316 | 1463 |
| | | 298 | 256 | 370 | 385 | 563 | 782 |
| | | 751 | 836 | 944 | 1030 | 1685 | 1903 |
| | | 819 | 962 | 1042 | 1293 | 2591 | 2980 |
| : DQN : Classical | | | | | | | |

**Fig. 9.13.** KPI for phase 2 deployment, comparing both the DRL controller and its classical reactive counterpart. Results are highlighted in green when DRL performances exceeds classical controls, and in red in the opposite case.

from the heat transfer and inertia of all other rooms, and (4) temperature continuously oscillates between set points, with lagged power calls and with timing at the end of the day to reach the larger dead band without falling into the discomfort zone, and while successfully avoiding electricity consumption during the expensive high price hours of the evening.

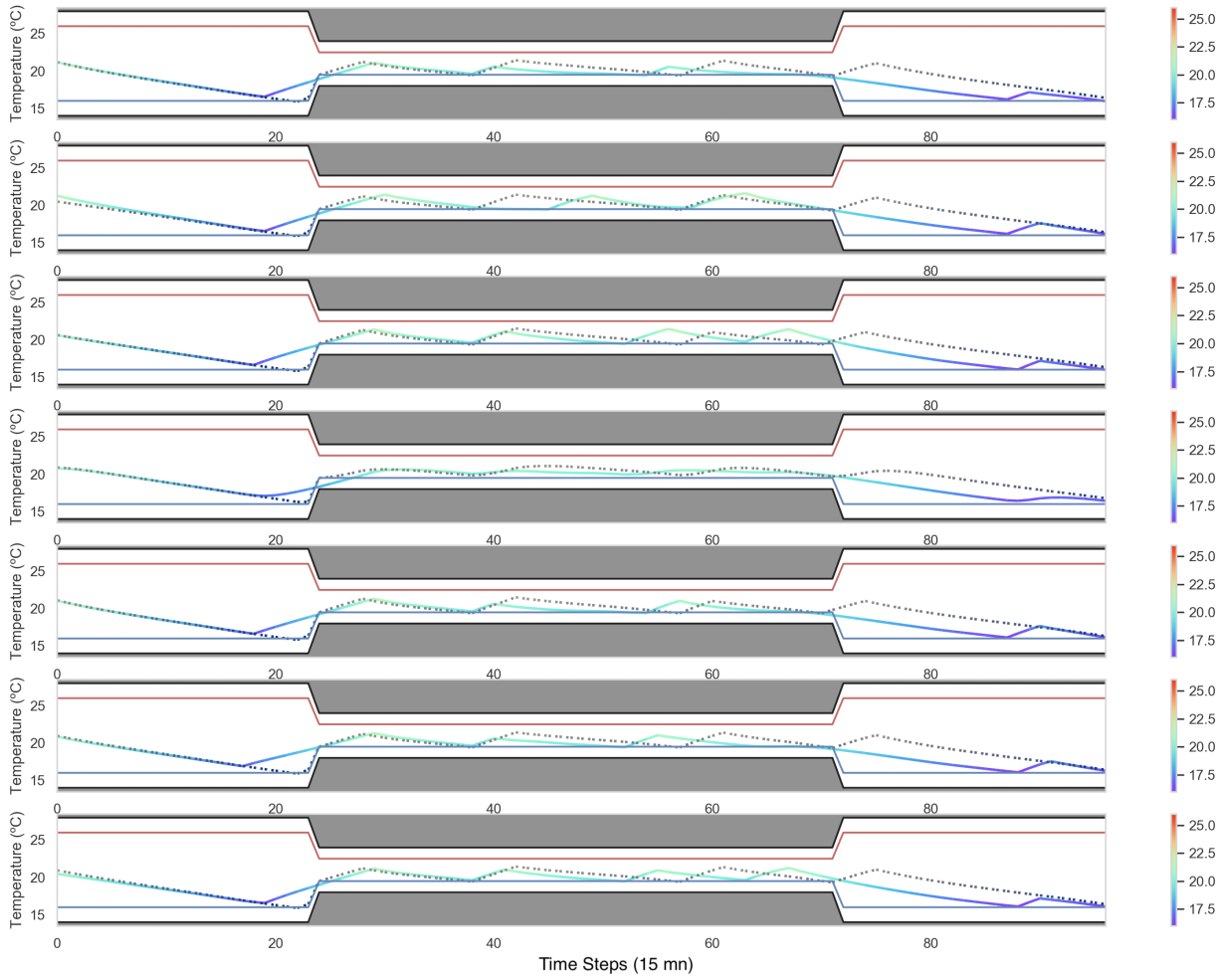


**Fig. 9.14.** Temperature variation in each zone for the small commercial center instance during a typical day in January 2020. The fully colored line represents the RL agent's result, while the grey dashed line depicts what a classical controller would have performed under the same conditions.

Just like figure 9.14 showed the hourly energy price awareness of the DQN controller, figure 9.15 illustrates an important flattening behavior in the power calls of the January 2020 month, and figure 9.17 an important peaks distribution shift toward lower values over the whole year. Such improvements are particularly important, as power-related costs usually account for the majority of the total expenses in buildings, and are prone to change

depending on different factors like geographic location and electrical operator contracts. This inherent variability makes a flexible solution like DRL an ideal approach to fit a broad range of specific eventualities by simply adapting the reward definition of the control agent.

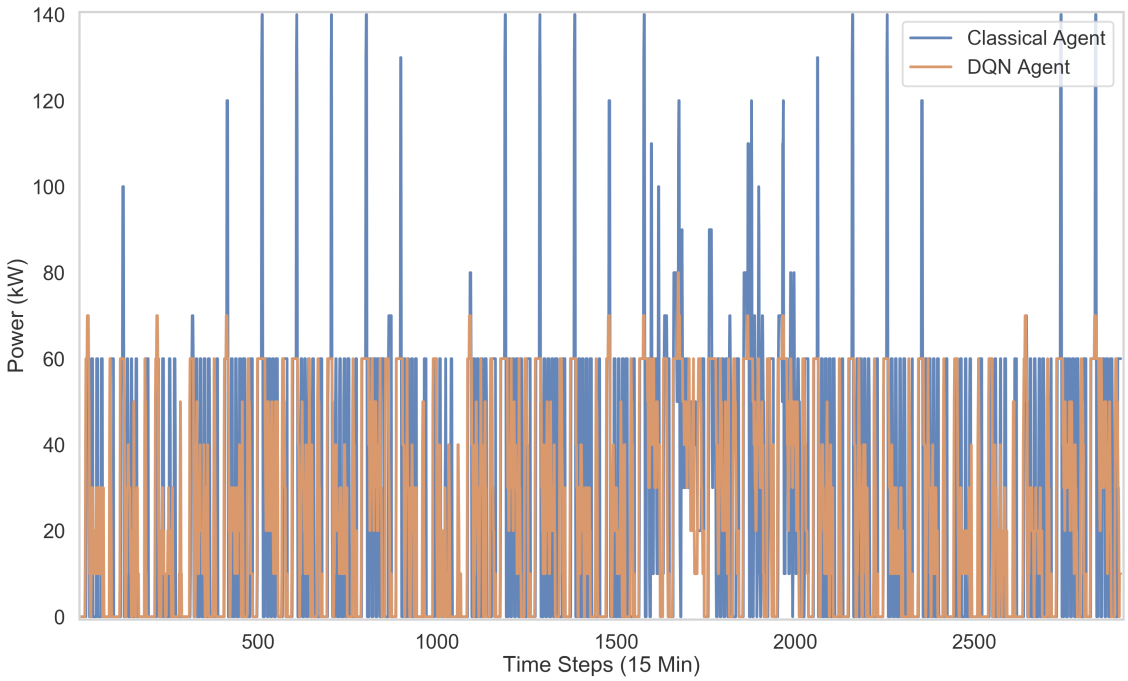


**Fig. 9.15.** Instantaneous 15 minutes power calls for phase 1 of instance C.

Despite being a competing objective to thermal comfort, equipment cyclability also shows enhancements over normal operations: for the largest instance, the number of cycles were reduced by 10% in Spring, 28% in Summer, 11% in Autumn, and 13% in Winter. The phenomenon can be directly observed in the control sequences comparison of instance B for a typical day in January 2020 (see figure 9.16).

Phase 2 results highlight a very important aspect, that is, the capacity of the DRL controller to cope well with all seasonalities and weather transitions. This aspect is usually challenging for this type of applications, making it a noteworthy advantage of the proposed methodology. The action set of the controller was never constrained or modified, and always had all the heating and cooling actions available. The agent autonomously learned how to blend efficiently heating and cooling in seasonality transitions, while properly focusing only on the important possibilities by itself in more extreme weathers. This suggests that while occasional retraining may be required during the first year with seasonality changes, an important autonomy could be reached with very sparse yearly updates once a complete year of data is obtained.

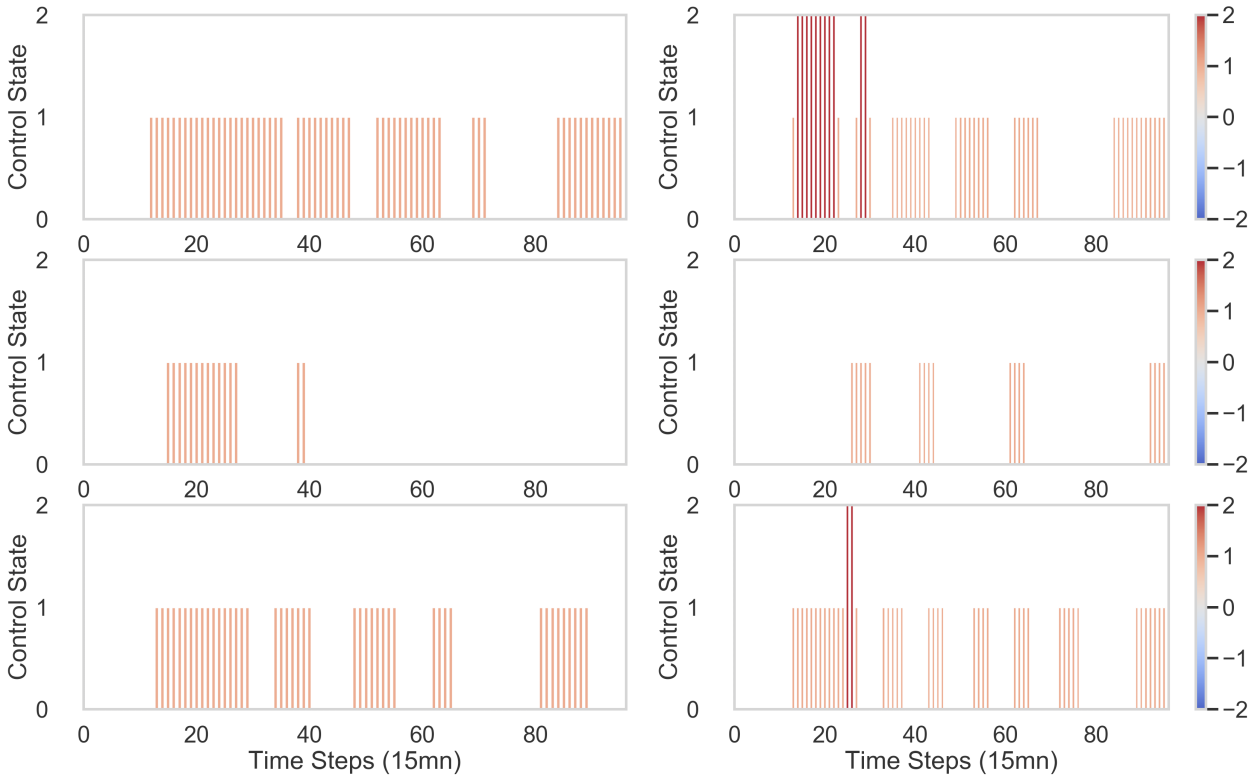


**Fig. 9.16.** 24-hour control sequences on instance B for the DQN controller (left) and classical controller (right) during a typical day of January 2020.

The training phase demonstrated an important continuous stability for all instances, with non-degrading and monotonically improving performance. Even after 80% of the epochs completed, pursuing training with a fixed exploration rate of $\epsilon = 0.05$ did not change significantly the actual approximated optimal Q-values following new neural network updates. Such behavior is important to monitor, as it suggests convergence toward a stable solution, and can even be used to halt training after the optimal number of training iterations for deployment.

Our general synthesis is that multi-system DQN controller deployment value increases with the complexity and opportunity potential of the considered instance. That is, systems exhibiting complex dynamics and transient behaviors which typically induce challenges to building operators and engineers, like fluctuating energy prices or contracts, a high number of simultaneous control components, dynamic building schedules, or user-defined set points, to name but a few. The *learning* and adaptive capability of DRL methodology then becomes highly profitable, and does not suffer from obsolescence in the long term. Applying this solution to smaller-scale buildings like houses or simpler systems still result in improvements, but can prove more impactful in higher volumes and with group coordination.

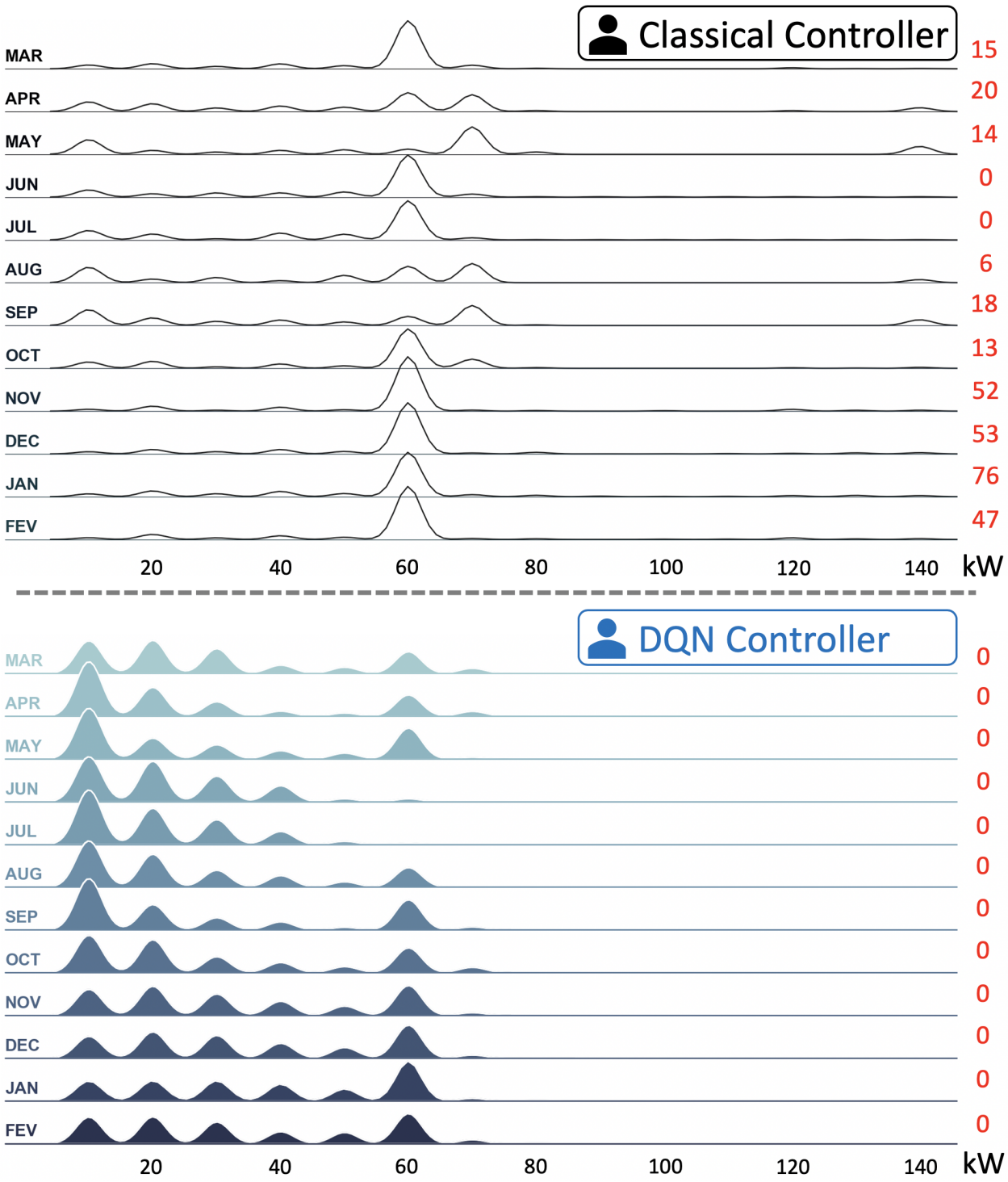


**Fig. 9.17.** Monthly non-zero instantaneous power consumption distributions for instance C, from March 15 2019 to March 15 2020. The red numbers on the right denote the number of power calls over 120 kW.

### 9.3.4. Real Application and Indirect Reinforcement Learning

Applying reinforcement learning on a real building instance requires a virtual twin, or at least a proper environment model, since the poor performance of the training phase can negatively affect human occupants' well-being and safety. With this in mind, we proceeded to a quick indirect reinforcement learning assessment by using ninety days of simulated data from a classical controller to train a deep learning model on the environment's transition dynamics (temperature difference in the present case). That is, we used experience generated by a standard reactive controller to train an approximate environment's transition model, to be able to learn an independent optimal policy using our afore-described DQN algorithm but without affecting the environment during the learning phase. Despite a slight (expected) loss in performance over the direct RL application, to our surprise, we are still able to outperform a traditional controller with respect to all the objectives. This can probably be explained by the fact that the only reward component relying on the model is the comfort. The rest, i.e. power, energy, equipement cycling and expenses are all deterministic and can be computed precisely for all the agent's actions.

In the light of our results, it is our belief that DRL and its extensions can cope with much more complex smart building instances, both in terms of scale and dynamics. Considering this, and leveraging the aforementioned temperature prediction framework, we are currently working on deploying our DRL method on real instances. The projection is to first demonstrate a more realistic indirect RL proof of concept on a large-scale realistic building simulation (using an external library from other research partners to do so) by the end of 2020; then to proceed with real instances in 2021 if everything works as expected.

In addition to its autonomous learning capabilities directly from raw observations of any entity, DRL reward definition can easily be tuned and adapted through a simple graphical user interface (GUI) post-implementation to reflect different optimization objectives. Furthermore, from a calculation perspective, a DQN controller presents the advantage of being light in deployment, as only matrix multiplications are required during the forward pass to access the policy. This can further be leveraged in the GUI to offer visualization and simulation tools as a transparency measure to building managers and operators.

# Chapter 10

# Control Distillation

Deploying statistical learning and control resources in large-scale production induces different challenges and opportunities compared to traditional research. One of the upsides of this reality, is the wealth and diversity of valuable content coming from different sources like engineering, applied research from different fields, empirical knowledge, etc. In order to take advantage of all the combined contributions, one needs to consider a proper equitable standardized framework to fuse efficiently and seamlessly all the benefits together.

In this perspective, we conclude our journey by presenting what we believe to be an elegant approach to combine and contrast multiple control resources in deployment, regardless of their origin. We finally close the chapter by giving an insight on planned future research and explorations to extend the material presented in the scope of this work.

## 10.1. Pareto Front and Multi-objective optimization

Multi-objective optimization[1] is the science of optimizing $k$ multiple, usually contending, objective functions $f_i(x),\ i \in \{1,2,...,k\}$:

$$\max(f_1(x),f_2(x),...,f_k(x)) \tag{10.1.1}$$

$$\text{subject to } x \in X\ , \tag{10.1.2}$$

where $X$ is the set of *feasible solutions*. The reader should note that while we chose to maximize the objective functions, the minimization equivalent follows exactly the same principle.

It is usually impossible for a given solution to optimize simultaneously all objective functions. Considering this reality, we introduce the concept of *Pareto optimal* solutions,

[1]Also often referred to as *multicriteria optimization*, *multi-objective programming*, *multiattribute optimization* or *Pareto optimization*

that is, solutions that cannot improve any objective without degrading at least one of the others. Furthermore, a solution $x_a$ is said to (Pareto) *dominate* another solution $x_b$ if:

(1) $f_i(x_a) \geq f_j(x_b)\ \forall i \in \{1,2,...,k\}$, and
(2) $f_j(x_a) > f_j(x_b)$ for at least one index $j \in \{1,2,...,k\}$.

The *Pareto frontier* (see figure 10.1) represents the set of *Pareto optimal* solutions $x^\star \in X$, i.e. solutions that are not dominated by any other feasible solutions.

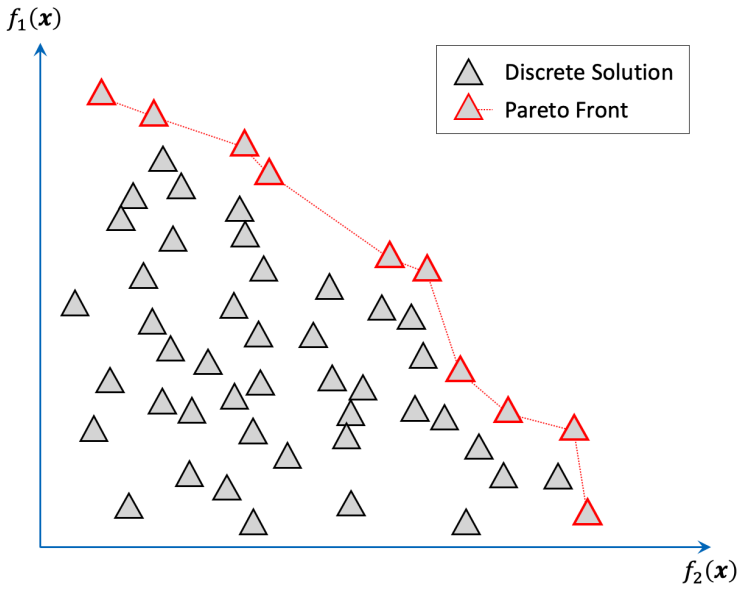


**Fig. 10.1.** Illustration of the Pareto front for discretized solutions with 2 objective functions to maximize.

## 10.2. State Compounding

In this section we formally introduce one of the important processes repeatedly used in the main *control distillation* interface, which we refer to as state *compounding*. The core idea of compounding is to take a control sequence, then to apply a high fidelity environment transition model with all available information to calculate precisely the trajectory of these specific controls. To reduce dimensionality, resulting states are then compressed into a representation where classical OR can be applied, yet augmented using all information obtained from the environment's predicted dynamics. The full process goes as follow:

(1) Starting from a control sequence $u^{(1)},u^{(2)},...,u^{(h)}$, where $h$ is the prediction *horizon*, the environment model is applied to predict the induced future states $x^{(1)}, x^{(2)}, ..., x^{(h)} \in \mathbb{R}^d$.

(2) These predicted states, usually of high dimension because of the model quality, are then projected on their primary component of interest, $x_1^{(i)}$ $\forall i \in \{1,2,...,h\}$, where the $x_j, j \in \{1,2,...,d\}$ represent the state vector components. In our smart building application, $x_1$ is the internal temperature measurement.

(3) The primary variable is then discretized into predefined states according to a meshing parameter $\Delta x_1$. In our case, the default value is chosen as the measuring equipment's uncertainty.

(4) The trajectory is finally represented as a deterministic dynamic program (DDP), as described in section 1.2, and can consequently be considered as a shortest path problem.

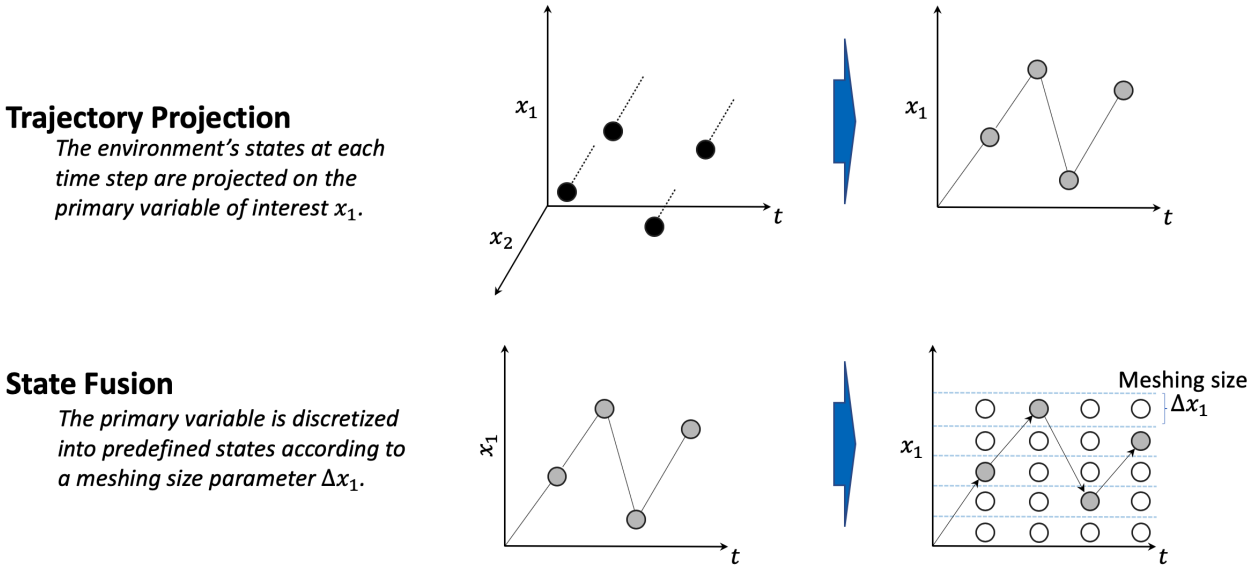


**Fig. 10.2.** Visual representation of the state compounding process.

## 10.3. The Control Distillation Interface

The control distillation interface is a mean to combine multiple proposed control sequences, comparing them on a fair basis, and fusing their optimal sub-components together to produce a globally optimal solution. In its most general form, the interface is initialized with the following inputs:

- A **baseline sequence** ${}^0U = \{{}^0U_1, {}^0U_2, ..., {}^0U_z\}$, where every control $U_i$, $i \in [1,2,...,z]$ is of the form $U_i = u_i^{(1)}, u_i^{(2)}, ..., u_i^{(h)}$, as introduced previously, and where $z$ is the number of sub-instances (zones). Typically, ${}^0U$ can be a safe and reliable solution like the default classical controls.

- **Individual (separable) additive objective functions** $\kappa_{ij}(U_i)$, $j \in \{1,2,...,k\}$, $i \in \{1,2,...,z\}$. These can be optimized locally with a guarantee of improving the global solution (for example, energy consumption) regardless of the other decisions.
- **Globally-dependent objective functions** $\lambda_m(U)$, $m \in \{1,2,...,l\}$, that depend on *all* the chosen controls (e.g. power price based on combined highest peak of the month) and that cannot be improved locally without complete knowledge of the whole sequence (optional).
- A **set of constraints** $\Gamma$, which can be refined according to each specific objective function (optional).
- A **set of unallowable sequences** $\Phi$, due to application constraints (optional).

From these inputs, the baseline solution is defined as the *deployment sequence* or *deployment solution* $U_{\text{deployment}}$, i.e. the sequence to send to the controller when a decision is required. The different objective functions are then calculated for this solution, to be used as reference to quantify subsequent improvements. The whole initialization process is depicted in the figure 10.3.

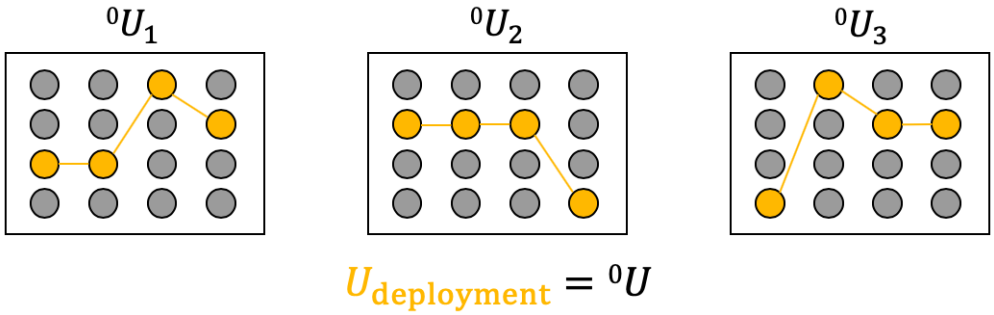


**Fig. 10.3.** Example of the Distillation interface initialization, for 3 sub-instances, where the baseline sequence $^0U$ is defined as the first deployment sequence $U_{\text{deployment}}$.

Our optimization problem consequently takes the form:

$$\max F = \max_{U} \sum_{i=1}^{z} \sum_{j=1}^{k} \alpha_{\kappa_{ij}} \kappa_{ij}(U_i) + \sum_{m=1}^{l} \alpha_{\lambda_m} \lambda_m(U) \ ,$$

subject to $\Gamma$ and $\Phi$.

where $F$ is our objective function, and the $\alpha$ are the weight coefficients of the different sub-objectives.

Once properly initialized, one of the main methods of the Distillation interface is the *control sequence injection*, taking one or multiple control sequences as input regardless of

their origin[2]. These sequences are then *compounded* (see previous section) and aggregated to the DDP problem of each sub-instance [3].

As a set of new sequences is injected, an *improve solution* method can be executed (repeatedly) to improve the control sequence of each sub-instance, and adapted whether parallel processing is available or not. The main idea behind this method is to recombine the optimal sub-paths of the different injected solutions to create the global optimal one[4]. Starting with the easiest scenario where $\lambda = \emptyset$, since the compounding process already expressed all the trajectories under a standardized DDP format, a simple shortest path algorithm like label-correcting methods, forward chaining or backward induction can directly be applied, resulting in an optimal solution at the sub-instance level. Specific constraints from $\Gamma$ and $\Phi$ are tracked during this process, cutting non-compliant branches or making their objective function value equal to $-\infty$ to avoid their consideration. All the aforementioned operations can be easily performed with multi-processing to speed up calculation, as each problem is independent of the others and represents an additive term in the total objective sum. The combined global deployment solution $U_{\text{Deployment}}$ is then updated with the new optimal paths (if any).

If $\lambda \neq \emptyset$ and globally-dependent objective functions are considered in the problem, then more caution needs to be taken, as each sub-instance cannot simply be optimized individually (at least not directly without introducing new mechanics). In that case, the simplest method is that either: (1) the same procedure as before can be applied by iterating sequentially over each sub-instance and considering the rest of them static, or (2) if multi-processing is available, the objective function gain over the old deployment sequence can be calculated for each sub-instance in parallel, before updating the most promising one. It is important to note that this approach is a lot more computation heavy, and performs poorly if the nature of the problem induces high objective function values only for specific combined controls. Of all the content presented in this chapter, this aspect is probably one of the biggest performance bottlenecks and the most prone to improvement in our proposed solution.

[2]It is however interesting to add a *signature* on the origin of the controls, to keep track of the most performing contributing methodologies.

[3]It is worth nothing here that an input sequence can be for only a specific sub-instance.

[4]The reader should be aware that we are combining perfect information policies to create a solution to a problem with uncertainty. Consequently, there is not necessarily any optimization guarantees in the real environment. Nevertheless, we argue that the quantity of information extracted from the high-fidelity model is directly proportional to the performance of the conversion.

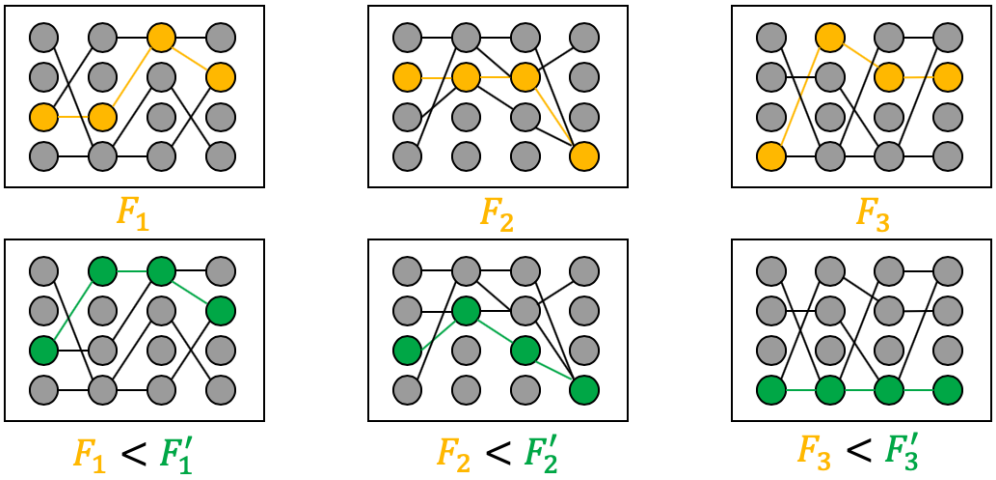


**Fig. 10.4.** (Top) Illustration of the *control sequences injection* method in the Distillation interface, with the original deployment sequence highlighted in yellow. (Bottom) Application of the *improve solution* method for the $\lambda = \emptyset$ case, where the $F$ and $F'$ represent respectively the original deployment sequence objective function values, and the new shortest path one. The reader should note that new optimal solutions were found for each sub-instance in this example, but that may not always be the case.

## 10.4. Deep Reinforcement Learning Driven Distillation

Just like *DeepMind* leveraged hundreds of hours of statistical learning in only a few seconds in their adapted MCTS (see Chapter 6), our objective is similarly to use all the information learned by a DRL agent during deployment. In our case, we however face the additional challenges of: 1) validating the control sequence's viability as no deployment mistakes can be made for the safety of individuals inside the environment, and 2) trying to find a global solution combining the optimal sub-components of a set of multiple control sequences from different origins.

As we demonstrated in the previous chapter, DRL performs very well in a shared parameters setting for global optimization with multiple simultaneous control instances. To this end, we expand each DRL "tail" $z$ (for "zone" in our application) by calculating the usual algorithm's output[5] but with respect to: (1) each individual objective functions $f_i(s,u)$, $i \in \{1,2,...,k+l\}$, and (2) pre-defined *weighted combined objective functions*[6] of interest of the form $\sum_{i=0}^{k+l} \alpha_i f_i^z(s,u)$. The main idea here is to use the latter to produce a strong initial baseline solution for our Distillation interface, and the former as a driver to guide the suggestion of new control sequences toward a specific objective, while minimizing

[5]The usual DRL output is either a state-action value function, an advantage or a probability distribution (policy). See chapter 6 for more details.
[6]We remind to the reader that $f_i$ depends on $s$ and $u$ here, because of the DRL method.

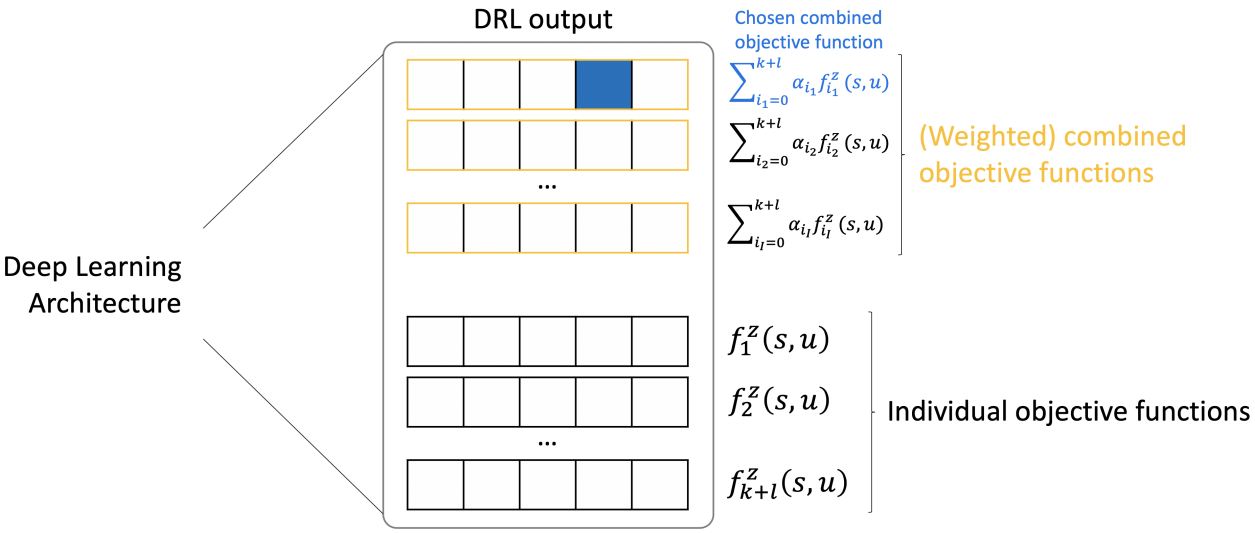


**Fig. 10.5.** Illustration of the quantities predicted by a DRL algorithm with DRL-driven Distillation: each tail is augmented to produce either a value function, an advantage or a probability distribution with respect to the function illustrated on the right. One of the weighted combined objective functions is then going to be used to initialize the baseline solution of the Distillation interface, and the individual objective functions to build a Pareto frontier to guide improvements.

performance losses in others.

Once the selected baseline actions are used to initialize the interface[7], according to the *argmax* of the chosen combined (weighted) objective function, all other external control sequences are injected and optimized. From this point, the user has a quantitative measure of the best available global solution, with respect to the main function $F$ and each individual objective functions. Looking back at the upgraded DRL, a Pareto frontier is created in the space of its outputs with respect to each individual objective functions. The actual resulting deployment sequence $U_{\text{deployment}}$ is then identified as the starting point of a "Pareto frontier walk" towards the closest Pareto-optimal action increasing the desired objective function[8]. This closest action is then submitted to the interface, and the process is repeated (the desired function to improve may change). This idea is in part similar to the Simplex algorithm which travels along the vertices of the optimal solutions polyhedra, but also to the shooting methods in some way, as only a parameter is ajusted before sampling and assessing a new trajectory.

[7]In practice it can be convenient to include both the DRL and a classical default solution to validate statistical learning.
[8]If the initial solution is not on the Pareto frontier, then the first returned solution is the closest one lying on the Pareto frontier.

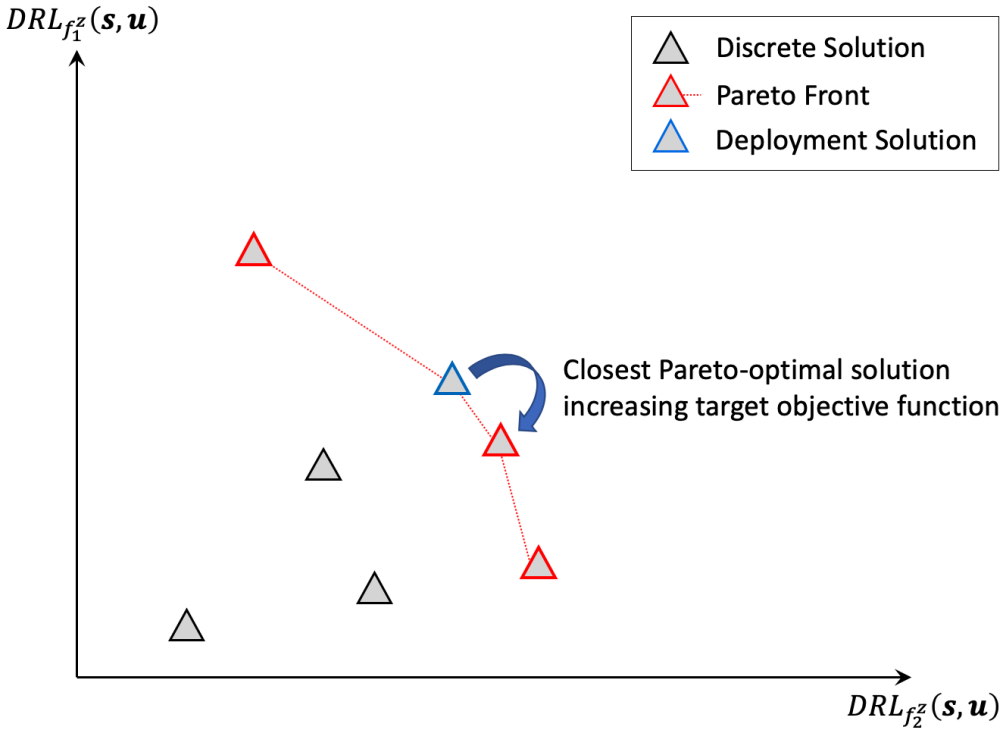


**Fig. 10.6.** Visual representation of "Pareto walking" from the currently considered deployment solution to the nearest Pareto-optimal one increasing the target objective function.

While the proposed methodology presents no rigorous guarantees, combining it with the distillation interface adds a safety layer that allows new proposed solutions to be ignored if they do not improve the actual deployment one.

# Chapter 11

# Conclusion and Research Avenues

Modern engineering usually involves an important unified contribution from various disciplines, in order to deliver the state-of-the-art performances that one is typically used to encounter in its everyday life. In this regard, we believe the work presented throughout this thesis to be a successful combination of control theory, statistical learning, physics and application-specific knowledge, which despite being only in its early stages, reached a concrete society-impacting level. Building on modern deep learning, deep reinforcement learning and physics content, we successfully applied and developed material for the smart grid's environment. While the specific energy storage application had to be aborted for financial reason, the subsequent work accomplished in smart buildings operation and control is the starting point of many promising developments beneficial to mankind.

While some interesting results arose from this work, which also led to many valuable lessons and precepts, in our perspective everything presented up to now represents nothing but a mere exordium. Much as the near-infinite future explorations space is a reality we acknowledge, we identify three major research axis we wish to develop concurrently in a near horizon, based on our interests:

- **Deep Learning**: given sufficient data, deep learning is probably to this day the most powerful function approximator available to mankind. Considering this aspect, while what we would call classical deep learning (like the related content we introduced in the scope of this work) results in good performances, we believe that the future of efficient and scalable applied artificial intelligence relies on the very recent advances in the deep learning field. To name a few examples, meta-learning or few-shots learning, multi-task learning, Attention-based interfaces such as neural Turing machines and many other modern developments are in our perspective the solution to the training burden and out-of-distribution generalization problem of

neural networks. Even the concept of Attention in its most crude form is of capital importance, as it can not only be used for artificial intelligence purposes, but can also prove powerful in control, or with physical quantities to blend domain knowledge with human interpretability and statistical learning.

- **Reinforcement Learning and Control**: the major advantage we see in developing deep reinforcement learning, is the fact that all the concomitant deep learning advances directly impact the performance and applicability of our control arsenal. Furthermore, like we mentioned in Chapter 6, (deep) reinforcement learning is the subject of very intensive research with exciting novelties arising on a regular basis. Nevertheless, despite some very interesting concepts like *curiosity*, or simply new state-of-the-art algorithms to consider for implementation, an important aspect we already highlighted is the importance to take a deeper look at all the work done by the control and operations research community, to make parallels and complement the related material in the artificial intelligence literature. To us, it is this equilibrium between novelty awareness and classical literature knowledge that is the key to a strong, efficient and minimalist optimal control infrastructure.

- **Physics and Engineering**: in conjunction to all aforementioned elements, we are convinced that the injection of physics insight is the secret to the next generation of applied artificial intelligence. By *physics*, we not only include empirical and theoretical domain-specific knowledge, but also fundamental quantities, principles and theories such as entropy, relativity, and many other Universe dynamics which can hardly be learned from a dataset as they are the result of thousands of years of combined human intelligence. Incorporating physics is also a medium for human comprehension and interpretation in the behavior resulting from the statistical learning process.

It is in this perspective that we conclude the final chapter of this volume, with the promise of a new tome aimed at swarm intelligence and multi-agent optimization in the forthcoming years. Through our future work we hope to reach an impactful contribution in the field of intelligent communities, with a keen focus on the smart grid application, and with the general aspiration of working for the greater good of Humanity.

# Part 4

# ARTICLE

# First Article.

# Autonomous Control in Smart Buildings: a Deep Reinforcement Learning Approach

by

Ysaël Desage[1], François Bouffard[2], Fabian Bastin[3], and Jean-Simon Venne[4]

([1]) Département d'Informatique et de Recherche Opérationnelle. Université de Montréal, Montreal, QC H3T 1J4, Canada

([2]) Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 0E9, Canada

([3]) Département d'Informatique et de Recherche Opérationnelle. Université de Montréal, Montreal, QC H3T 1J4, Canada

([4]) BrainBox AI Labs, 2075 Boulevard Robert-Bourassa, Montreal, QC H3A 2L1, Canada

This paper was submitted in IEEE, in the category *Transactions on Smart Grid*, and published as a cahier du GERAD.

My main contributions are:

- Paper writing;
- Implementation and application of the proposed methodology;
- Implementation and application of the presented simulation.

The 3 coauthors revised the paper thoroughly, providing very detailed feedback and suggestions with respect to their own field of expertise.

Résumé. L'apprentissage profond a redéfini les normes modernes et la performance dans les domaines de la vision informatique et du traitement du langage. Avec l'accroissement des données provenant des infrastructures de mesure de l'énergie dans les réseaux électriques, nous assistons à un foisonnement des occasions d'optimisation dans ces réseaux. Nous considérons donc ici le problème de commande optimale des systèmes de chauffage, climatisation et ventilation d'un bâtiment intelligent avec une infrastructure de calcul à basse puissance et ne nécessitant que peu de communications. Pour ce faire, nous introduisons une méthode de commande autonome, multi-système basée sur l'apprentissage par renforcement profond. Malgré l'application de plusieurs mesures assurant un niveau de service thermique minimal, nous démontrons des améliorations notoires en termes de coûts d'énergie, de puissance, de confort thermique et de nombre de cycles en utilisant une adaptation de l'approche de Q-apprentissage profond sur des cas d'espèce basés sur des simulations de modèles physiques calibrés sur des données metéorologiques historiques. Nous quantifions les résultats de l'optimisation et illustrons sa flexibilité et comment cette méthode peut facilement être étendue à des édifices de plus en plus grands en comparant sa performance avec les contrôleurs classiques réactifs.
**Mots clés :** Bâtiments, Apprentissage Profond, Apprentissage par Renforcement, Consommation Énergétique, Contrôle, Optimisation, Appels de Puissance, Réseaux Intelligents

Abstract. Deep learning has redefined modern standards and performance in several areas such as computer vision and natural language processing. With increasing amounts of frequently sampled data in advanced metering infrastructure, similar opportunities are readily available for smart grid actors' optimization. In this regard, we consider the problem of remote high-granularity control with low computational power in deployment and intermittent connectivity for heating, ventilation, and air-conditioning components in smart buildings. Thereupon, we introduce an adapted autonomous multi-system command infrastructure based on deep reinforcement learning. Through several deployment safety measures, we demonstrate significant improvements in expenses, thermal comfort, energy consumption, power peaks and equipment cycling using an adaptation of the Deep Q-Learning algorithm on case studies of physics-based simulations relying on real historical weather data. We quantify the resulting optimization and illustrate both the scalability and flexibility of our approach by comparing the trained controller to its classical reactive counterparts on instances requiring simultaneous control on up to seven parallel systems.
**Keywords:** Buildings, Deep Learning, Deep Reinforcement Learning, Energy Consumption, Optimal Control, Optimization, Power Consumption, Smart Grid

# Autonomous Control in Smart Buildings: a Deep Reinforcement Learning Approach

Ysaël Desage, François Bouffard, *Senior Member, IEEE,* Fabian Bastin, and Jean-Simon Venne

***Abstract*—Deep learning has redefined modern standards and performance in several areas such as computer vision and natural language processing. With increasing amounts of frequently sampled data in advanced metering infrastructure, similar opportunities are readily available for smart grid actors' optimization. In this regard, we consider the problem of remote high-granularity control with low computational power in deployment and intermittent connectivity for heating, ventilation, and air-conditioning components in smart buildings. Thereupon, we introduce an adapted autonomous multi-system command infrastructure based on deep reinforcement learning. Through several deployment safety measures, we demonstrate significant improvements in expenses, thermal comfort, energy consumption, power peaks and equipment cycling using an adaptation of the Deep Q-Learning algorithm on case studies of physics-based simulations relying on real historical weather data. We quantify the resulting optimization and illustrate both the scalability and flexibility of our approach by comparing the trained controller to its classical reactive counterparts on instances requiring simultaneous control on up to seven parallel systems.**

***Index Terms*—Buildings, Deep Learning, Deep Reinforcement Learning, Energy Consumption, Optimal Control, Optimization, Power Consumption, Smart Grid.**

## I. Introduction

ENERGY consumed in the buildings sector, both residential and commercial, accounts for near 40% of the total worldwide energy consumption [1], and beyond 30% of $CO_2$ emissions [2]. Considering that 230 billion square meters of new construction is expected over the next 40 years [3], such numbers make smart buildings one of the major actors in the modern power grid's infrastructure.

From this perspective, in this paper we consider the multi-objective problem of optimizing expenses, thermal comfort, power peaks, energy consumption and equipment cycling in smart buildings equipped with multiple air-handling units (AHU) such as Roof Top Units (RTU), and connected to the electrical grid. We assume control will be applied with high granularity (decision epochs every 15 minutes), from a remote low-computational power device having local access to all the heating, ventilation and air-conditioning (HVAC) components inside the building (see Fig. 1). For application realism purposes, we also consider sparse intermittent connectivity with the remote control device, making it independent from the central computing source except for occasional data transfers. Finally, we impose a set of pre-defined fixed constraints on thermal values for the safety of the users inside the buildings, which will automatically trigger a fall-back position to classical controls if needed.

This work was supported in part by Mitacs and by BrainBox AI labs.

Y. Desage and F. Bastin are with the Departement d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montreal, QC H3T 1J4, Canada (e-mail: ysael.desage@me.com; bastin@iro.umontreal.ca)

F. Bouffard is with the Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 0E9, Canada and with the Groupe d'études et de recherche en analyse des décisions (GERAD), Montreal, QC H3T 1J4, Canada (email: francois.bouffard@mcgill.ca).

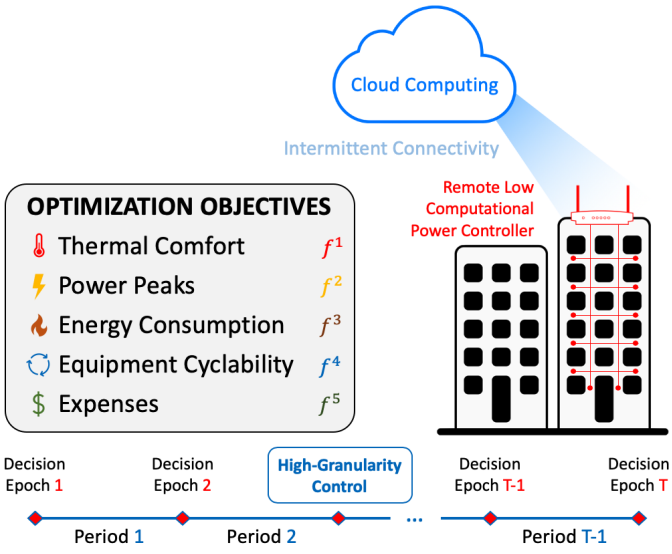


Fig. 1. Remote high-granularity control problem considered.

The considered multi-objective optimization problem can be expressed as a single objective problem using weighted contributions of the form:

$$\max \sum_{t=1}^{T} \left\{ \alpha_t^1 f_t^1 + \alpha_t^2 f_t^2 + \alpha_t^3 f_t^3 + \alpha_t^4 f_t^4 + \alpha_t^5 f_t^5 \right\} , \quad (1)$$

where $t$ indexes time up to a horizon $T$, and $f_t^i$ and $\alpha_t^i$, $i \in \{1, 2, ..., 5\}$, are the different objective functions as depicted in Fig. 1, and their respective weighting coefficients. This optimization is subject to the thermodynamics of the building, the occupants' safety measures, and the technical specificities of the HVAC equipments.

Numerous approaches have been proposed in literature to address smart building control, including dynamic programming [4], [5], model-predictive control [6], and machine learning [7], [8], to name but a few. While demonstrating successful results, such methods are either very compositionally expensive in deployment, require continuous two-way communication protocols, imply numerous hours of building-specific engineering, or simply do not scale up properly to large instances.

In contrast, our proposed deep reinforcement learning solution leverages the rich amount of temporal observations through a specialized recurrent deep learning architecture,

resolves the important exponential action space growth of larger parallel control instances, and allows an efficient global optimization through a shared parameter setting. Lastly, our solution can be easily deployed in its full potential requiring only a few matrix multiplications on the deployment device.

## II. Reinforcement Learning

Reinforcement Learning (RL) is one of the three main machine learning (ML) paradigms where training information is used to evaluate actions (rather than instruct), in order to maximize a numerical reward signal defined in accordance to a specific goal [9]. RL relies on the framework of Markov Decision Processes (MDP), where an agent $a$ undergoes continuous or episodic interactions with its environment. At each step of a sequence of discrete time steps $t$, up to a horizon $T \in [0, \infty)$, the decision maker receives a partial observation $o_t \in \mathcal{O}$ of the real state $s_t \in \mathcal{S}$ the environment $E$ is currently in, where $\mathcal{O}$ and $\mathcal{S}$ represent the discrete finite sets of observations and states, respectively. Based on the perceived observation, the controller then applies its policy $\pi$ to choose a control $u_t$ from a set of allowable actions $\mathcal{U}_s$[1], which triggers a transition of the environment according to the probability function $P_t(s_{t+1}|s_t, u_t)$ into a new state $s_{t+1}$ and returns a new (partial) observation $o_{t+1}$ along with a reward $r_t \in \mathbb{R}$. The objective of the decision maker is to maximize the expected cumulative sum of such reward, called the *return*:

$$R_t = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t(s_t, u_t)\right] \quad , \tag{2}$$

where the discounting factor $\gamma \in [0, 1)$ accounts for the desirability of short versus long-term rewards. From this, we define the $Q$-function (or $Q$-factors) [10] of a given policy $\pi$ as the total expected return from being in state $s$, applying action $u$ and thereafter following policy $\pi$:

$$Q^\pi(s, u) = \mathbb{E}\big[R_t | s_0 = s, u_0 = u, \pi\big] \quad . \tag{3}$$

It is common to use a function approximator to estimate this or other quantities in RL. When a neural network is used for this purpose, we typically use the term *deep reinforcement learning* (DRL).

### A. Deep Q-Learning

The optimal $Q$-factors can be defined as [11]:

$$Q^\star(s, u) = \max_\pi \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t(s_t, u_t) | s_0 = s, u_0 = u\right] \quad , \tag{4}$$

which obey an important recursive relation known as the *Bellman Equation*: starting from the intuition that if the optimal value $Q^\star(s', u')$ of the state $s_{t+1} = s'$ at the next time-step was known for all possible actions $u_{t+1}$, then the optimal strategy is to select that action $u'$ maximizing the expected value of $r + \gamma Q^\star(s', u')$. Algorithms of the classical *Q-Learning* [12] family leverage this principle and convert the Bellman equation into an iterative update of the form

$$Q_{i+1}(s_t, u_t) = \mathbb{E}_{s_{t+1}}\big[r + \gamma \max_{u_{t+1}} Q_i(s_{t+1}, u_{t+1}) | s_t, u_t\big] \quad . \tag{5}$$

The *Deep Q-Network* (DQN) [13] algorithm uses a deep neural network as a function approximator to predict $Q(s, u; \theta) \approx Q^\star(s, u)$, where $\theta$ denotes the parameters of the network. These parameters are updated periodically as a supervised learning task using a mean-squared error loss $\mathcal{L}$ of the difference between the target $y = r + \gamma \max_{u'} Q(s', u'; \theta_i^-)$ and the old estimate $Q(s, u; \theta)$:

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,u,r}\Big[\big(y - Q(s, u; \theta_i)\big)^2 + \mathbb{E}_{s,u,r}\big[\mathrm{Var}'_s(y)\big]\Big] \quad , \tag{6}$$

where $\theta_i^-$ and $\mathbb{E}_{s,u,r}\big[\mathrm{Var}'_s(y)\big]$ represent the network's parameters from a previous iteration and the expected variance of the target, respectively. Crude Monte Carlo estimates, or Sample Average Approximation (SAA) are then typically used in conjunction with incremental gradient methods to converge to a solution.

Lastly, [13] also introduces *experience replay* to stabilize learning and smooth out the training distribution, which consists of storing experience tuples $e_t = (s_t, u_t, r_t, s_{t+1})$ in a memory buffer. During the inner training loop of the algorithm, training examples are then randomly selected and mini-batch Q-learning updates are performed with respect to (6). This characteristic makes the DQN an *offline* algorithm, where the agent learns a different optimal policy from the one used to act in the environment during training.

## III. Methodology

Given the sequential nature arising from time dependency in the partial observations $o_t$, we consider a sequence-adapted bidirectional Long Short-Term Memory (LSTM) architecture as function approximator for the DRL (details are illustrated in Fig. 2). We refer the interested reader to Appendix A for a more thorough presentation of essential deep learning concepts.

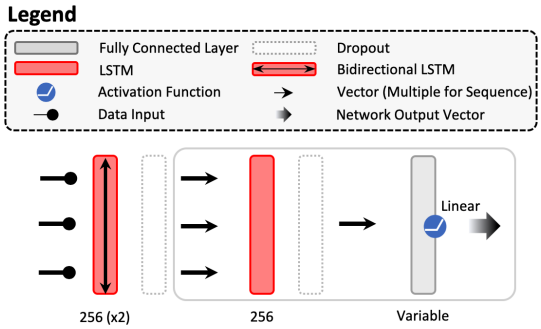


Fig. 2. Deep bidirectional LSTM architecture used as function approximator. The grey squares represent the duplicated components for each independent Q-function output. Activations are not explicitly shown for the LSTM layers.

From the reinforcement learning perspective, we extend the classical DQN framework by implementing its double Q-learning variant [14]. We then use the aforementioned neural network architecture to directly map observation sequences $\{o_{t-H}, ..., o_{t-1}, o_t\}$ to $Q$-values, where $H$ denotes the *observation history* considered by the agent[2]. Moreover, we further

[1] It is typical in RL to consider the same set of actions $\mathcal{U}_s = \mathcal{U}\ \forall s \in \mathcal{S}$.

[2] To rigorous intents, $\{o_{t-H}, ..., o_{t-1}, o_t\}$ can be considered as the DRL agent's *state* in itself, to respect the Markovian property.

leverage the deep learning infrastructure by considering not only a single $Q$-value vector for all the possible actions like in the original DQN algorithm, but rather parallel $Q$-value output streams for every present control system (see Fig. 3). This can be seen as a simplistic multi-agent framework adaptation, as it allows both an important action space factorization, and the possibility to provide system-specific actions and granularities for each controller. It is also computation-efficient because it requires only a single forward pass calculation, while still benefiting from shared parameters which offer a global generalization and optimization scheme.

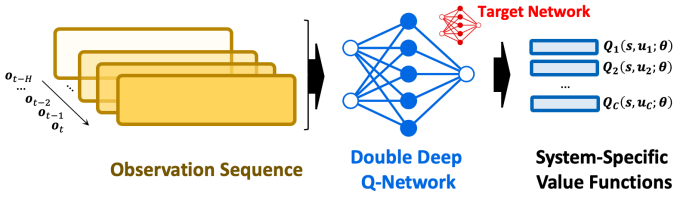


Fig. 3. Proposed Double Deep Q-Network using time series adapted deep learning architectures to map a sequence of observations $\{o_{t-H}, ..., o_{t-1}, o_t\}$ to individual parallel $Q$-value streams, one for each of the $C$ control systems.

Despite the high potential for performance exploration and tuning, we purposely fix the hyper-parameters of the DRL algorithm with the objective of testing application robustness and flexibility in a broad range of different instances, while reducing the associated computational burden. This also useful for assessing deployment autonomy, as no post-implementation interventions are performed.

## IV. Case Studies

### A. Case Studies' Setup

We perform our proof of concept through building simulations by accounting the interaction of all the major thermodynamics actors in the system: shared walls and doors between zones, open spaces and room content, external temperature, users activities, solar radiance, and HVAC components behavior. Real historical weather temperature is used to drive the boundary conditions of the simulation, internal thermal properties are chosen to represent archetype building types and characteristics, and the HVAC systems considered are RTUs with two heating stages and two cooling stages for a total of five possible control indexes if we include the "off" position. Finally, human activities and solar contributions are sampled from normal distributions with parameters varying depending on each building's nature and schedule.

For building types, we consider the three following distinct instances: a house (or an apartment) with two AHUs consuming 10kW for stage 1, and 15kW for stage 2; a retail store with three similar but larger systems having a 20kW stage 2; and finally a small commercial center with seven independent controllers identical to the retail store. We assume a 90% efficiency on all equipment, meaning that 90% of the input power is converted into heating or cooling. Each building instance (see Fig. 4) is represented by its own thermal circuit (the reader can refer to the Appendix B for more technical thermodynamics and modeling details) and includes heat contributions from users, solar radiance and HVAC components in each individual zone (see Fig. 5 for simulation parameters). The comfort zone is defined to be between 19.5°C and 22.5°C, but while it is kept constant in instance A, set points are scheduled in the two other instances to adjust the dead band between 16°C and 26°C during the non-occupancy hours, from 6 PM to 6 AM. The unitary kWh electricity price varies during the day, taking the value of 0.132$ from 7AM to 11AM, 0.095$ from 11AM to 3PM, 0.132$ from 3PM to 7 PM, and 0.05$ in the remaining hours. Power is charged based on the month's highest peak, at the rate of 14.58$/kW. While a realistic simulation is targeted, our main objective is also to introduce a high level of variation and uncertainty, to illustrate the generalization and adaptation capacity of our DRL controller.

To increase application realism, we perform training in a centralized fashion from a cloud computing resource, which then transfers the weights matrix to the local remote controller. During deployment, only the results of the forward pass are accessible to this device, with weight updates possible only through a pre-defined schedule. To assess performance on different deployment steps, with increasing amounts of available data, we divide the simulation into two phases:

- Phase 1: One continuous month deployment with an initial training on two months of similar conditions.
- Phase 2: Year-round deployment, with 8 months of initial training data and no weight update.

Phase 1 is evaluated on January 2020 (one of the coldest months) and trained from November 2019 to the end of December 2019, while phase 2 is trained from July 7, 2018 to March 14, 2019, then tested from March 15, 2019 to March 15, 2020 (see Fig. 6).

For the RL side of the simulation, the environment's partial observation includes the following information from the previous hour up to 15 minutes before the present time: temporal iteration's value $t$, and cyclic features of the form $\sin(2\pi t/T)$ and $\cos(2\pi t/T)$ where $T$ is the episode's length; the HVAC systems' index; internal and external temperature; temperature set points; energy consumed in each zone; power calls in each zone; and highest building power peak since the beginning of the month. Three actions are considered for each HVAC controller: heat (increases control index), do nothing (leaves index unchanged), or cool (lowers control index). Cooling stage 2 is represented by the lowest control index 0, while heating stage 2 is indexed by the maximum, 4. The reward process is built as follows: the agent receives a periodic reward of 50 at each time step while being between the temperature set points, and -20 if outside of the range, to stimulate comfort; a -1000 penalty each time the temperature is 2 degrees above or below the set points; a negative reward proportional to the power consumption each time the current highest power peak of the month is exceeded; a penalty linearly scaling to the energy cost at each time step; and finally a -15 reward for each action different from "do nothing", to reduce useless toggling. As a safety measure, the agent is replaced by classical controls until the temperature is back within comfort range when temperature exceeds or falls bellow 2 degrees of the set points. The resulting mathematical system for the agent is to maximize its return for a 24-hours period, while being exposed to all the combined reward mechanisms described above.

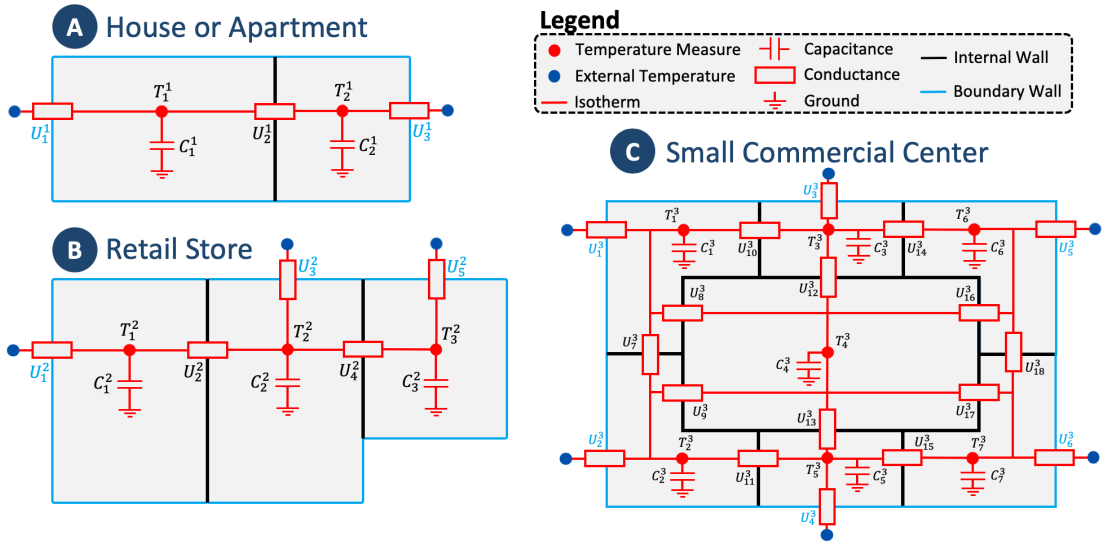


Fig. 4. Thermal RC circuit instances considered in the building simulation. Independent solar, human and HVAC heat contributions are added at every internal temperature measurement point, and the external temperature is provided from real historical data.

| | |
|---|---|
| **INSTANCE A** | |
| $U_1 = 300, U_2 = 100, U_3 = 250$ | W/K |
| $C_1 = 11 \times 10^6, C_2 = 11 \times 10^6$ | J/K |
| **INSTANCE B** | |
| $U_1 = 400, U_2 = 1000, U_3 = 200, U_4 = 750, U_5 = 300$ | W/K |
| $C_1 = C_2 = 12 \times 10^6, C_3 = 10 \times 10^6$ | J/K |
| **INSTANCE C** | |
| $U_1 = U_2 = U_3 = U_4 = U_5 = U_6 = 200$ | |
| $U_7 = U_{10} = U_{11} = U_{14} = U_{15} = U_{18} = 500$ | W/K |
| $U_8 = U_9 = U_{12} = U_{13} = U_{16} = U_{17} = 1000$ | |
| $C_1 = C_2 = C_6 = C_7 = 11 \times 10^6, C_4 = 12 \times 10^6$ | |
| $C_3 = C_5 = 10 \times 10^6$ | J/K |

Fig. 5. Value of the parameters used in the simulation.

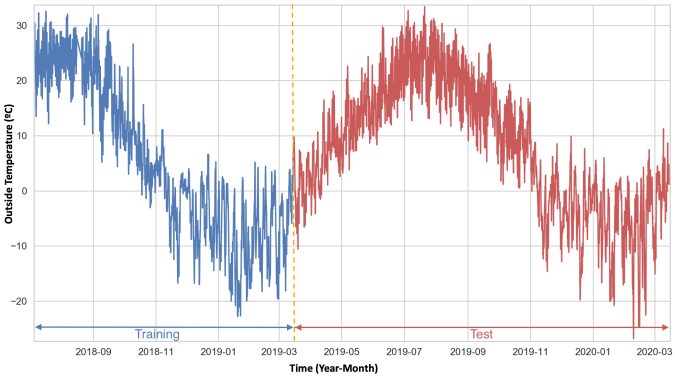


Fig. 6. Outside temperature in Montreal (QC), Canada, from July 7 2018 to March 15 2020. Data originates from the Dark Sky API [15], and the training data for phase 2 is depicted in blue, while the test deployment data is shown in red.

### B. DQN Training Parameters

The training phase of the modified DQN consists in 50 000 epochs of 24-hours episodes simulation, using an $\epsilon$-greedy policy where either a random action is chosen for every controller with probability $\epsilon$, or a stochastic policy using the vector of normalized $Q$-values as a probability distribution over the actions (sometimes referred to as *Boltzmann Policy* [16]) is applied instead. Using this methodology bolsters appropriate exploratory behavior in the long term, while still allowing convergence towards a final random optimal policy, if needs be (unlike the original algorithm always deterministically using the *arg max* of the Q-factors). Starting with $\epsilon = 1$, we perform a linear decrement to reach a floor value of $0.05$ at $80\%$ of the training epochs. Optimization is achieved with the *Adam* [17] gradient descent algorithm, with constraints to limit the gradient norm between -1 and 1 to avoid instability arising from a major update. Finally, a scheduled learning rate is applied with a starting value of $0.001$, reduced to $5 \times 10^{-4}$ at $30\%$ of training, then finally set to $1 \times 10^{-4}$ from $60\%$ to the end of the computation.

### C. Results and Discussion

We assess the test simulations by defining Key Performance Indicators (KPI) with respect to each initial target objective: total and individual costs for the expenses; average daily discomfort time for thermal comfort; average daily energy consumed for energy consumption; highest overall peak for power peaks; and total number of performed cycles for equipment cyclability. Following these definitions lead to the conclusion that the multi-objective optimization is successful overall, as the DQN controller outperforms or equals its classical reactive peer for all KPIs in phase 1 (see Fig. 7), and nearly all in phase 2 (see Fig. 8).

The only poorer performances in the year-round deployment are the ones related to discomfort and energy. It is however important to note that the difference between both controllers is smaller than the model's time step in the first case, making it negligible in the context of this simulation, while in the second case energy cost savings were still observed despite slightly more consumed energy. Running more training epochs or having a complete year of data would probably solve or improve these aspects. Lastly, it is worth mentioning from a safety perspective that security measures were not triggered at any point for phase 1, and only twice for phase 2 during

the coldest days of winter, illustrating the ability to operate reliably within strictly established constraints.

| Measure | Instance A | | Instance B | | Instance C | |
|---|---|---|---|---|---|---|
| Total Cost ($) | 841.8 | 865.7 | 1303.7 | 1331.6 | 1240.4 | 2266.2 |
| Power Cost ($) | 437.4 | 437.4 | 874.8 | 874.8 | 1020.6 | 2041.2 |
| Energy Cost ($) | 404.4 | 428.3 | 428.9 | 456.8 | 219.8 | 225.0 |
| Average Daily Discomfort Time (Minutes) | 31.5 | 106.25 | 43.2 | 108.7 | 39.2 | 91.1 |
| Average Energy Consumed Daily (kWh) | 158.3 | 167.2 | 168.9 | 176.9 | 87.4 | 89.0 |
| Highest Power Peak (kW) | 30.0 | 30.0 | 60.0 | 60.0 | 70.0 | 140.0 |
| # Equipment Cycles (Cycles) | 260 | 315 | 394 | 418 | 925 | 1038 |
| : DQN : Classical | | | | | | |

Fig. 7. KPI for phase 1 deployment, comparing both the DRL controller and its classical reactive counterpart. Results are highlighted in green when DRL performances exceeds classical controls, and in red in the opposite case.

| Measure | | Instance A | | Instance B | | Instance C | |
|---|---|---|---|---|---|---|---|
| Total Overall Cost ($) | Spring | 1301.1 | 1770.3 | 2050.8 | 2952.2 | 3246.9 | 6327.2 |
| | Summer | 951.6 | 1424.0 | 1414.5 | 2755.3 | 2682.5 | 4179.2 |
| | Autumn | 1713.7 | 1972.7 | 2690.8 | 3304.5 | 3356.2 | 6436.4 |
| | Winter | 2445.0 | 2510.3 | 3527.1 | 3897.2 | 4535.5 | 6749.0 |
| Total Power Cost ($) | | 874.8 | 1312.1 | 1603.8 | 2478.6 | 3061.8 | 6123.6 |
| | | 874.8 | 1312.2 | 1312.2 | 2624.4 | 2624.4 | 4082.4 |
| | | 1093.5 | 1312.2 | 2041.2 | 2624.4 | 3061.8 | 6123.6 |
| | | 1312.2 | 1312.2 | 2332.8 | 2624.4 | 3936.6 | 6123.6 |
| Total Energy Cost ($) | | 426.2 | 458.1 | 447.0 | 473.6 | 185.2 | 203.6 |
| | | 76.8 | 111.8 | 102.3 | 130.9 | 58.1 | 96.9 |
| | | 620.2 | 660.5 | 649.6 | 680.1 | 294.4 | 312.8 |
| | | 1132.8 | 1198.1 | 1194.3 | 1273.1 | 598.9 | 625.5 |
| Average Daily Discomfort Time (Minutes) | | 44.7 | 69.8 | 44.3 | 59.0 | 53.2 | 39.9 |
| | | 27.7 | 23.3 | 13.7 | 23.6 | 28.3 | 19.5 |
| | | 33.5 | 52.7 | 54.6 | 71.67 | 53.0 | 39.7 |
| | | 40.5 | 111.2 | 61.0 | 112.3 | 49.3 | 87.5 |
| Average Energy Consumed Daily (kWh) | | 54.2 | 56.5 | 57.8 | 56.9 | 25.1 | 24.7 |
| | | 9.2 | 11.9 | 11.8 | 13.9 | 6.5 | 10.0 |
| | | 149.8 | 156.9 | 85.0 | 85.0 | 39.7 | 40.0 |
| | | 150.1 | 159.8 | 159.9 | 168.0 | 80.6 | 83.8 |
| Highest Power Peak of the Season (kW) | | 20.0 | 30.0 | 40.0 | 60.0 | 70.0 | 140.0 |
| | | 20.0 | 30.0 | 30.0 | 60.0 | 60.0 | 140.0 |
| | | 30.0 | 30.0 | 40.0 | 60.0 | 70.0 | 140.0 |
| | | 30.0 | 30.0 | 60.0 | 60.0 | 100.0 | 140.0 |
| Total Equipment Cycles (Cycles) | | 704 | 729 | 879 | 918 | 1316 | 1463 |
| | | 298 | 256 | 370 | 385 | 563 | 782 |
| | | 751 | 836 | 944 | 1030 | 1685 | 1903 |
| | | 819 | 962 | 1042 | 1293 | 2591 | 2980 |
| : DQN : Classical | | | | | | | |

Fig. 8. KPI for phase 2 deployment, comparing both the DRL controller and its classical reactive counterpart. Results are highlighted in green when DRL performances exceeds classical controls, and in red in the opposite case.

Fig. 9 visually depicts a combination of important optimal HVAC behaviors autonomously learned by the DRL controller, which match suggested theoretical guidelines found in the HVAC literature [18], [19]: 1) temperature barely touches the set points and reacts pre-emptively just before doing so, 2) pre-heating in the morning is linear and with a precise phase shift, yet respects the more restrictive set point right on time during the occupancy schedule, 3) the central zone never triggers HVAC as it knows it will benefit from the heat transfer and inertia of all other rooms, and 4) temperature continuously oscillates between set points, with lagged power calls and with timing at the end of the day to reach the larger dead band without falling into the discomfort zone, and while successfully avoiding electricity consumption during the expensive high price hours of the evening.

Just like Fig. 9 showed the hourly energy price awareness of the DQN controller, Fig. 10 illustrates an important flattening behavior in the power calls of the January 2020 month, and Fig. 11 an important peaks distribution shift toward lower values over the whole year. Such improvements are particularly important, as power-related costs usually account for the majority of the total expenses in buildings, and are prone to change depending on different factors like geographic location and electrical operator contracts. This inherent variability makes a flexible solution like DRL an ideal approach to fit a broad range of specific eventualities by simply adapting the reward definition of the control agent.

Despite being a competing objective to thermal comfort, equipment cyclability also shows enhancements over normal operations: for the largest instance, the number of cycles were reduced by 10% in Spring, 28% in Summer, 11% in Autumn, and 13% in Winter. The phenomenon can be directly observed in the control sequences comparison of instance B for a typical day in January 2020 (see Fig. 12).

Phase 2 results highlight a very important aspect, that is, the capacity of the DRL controller to cope well with all seasonalities and weather transitions. This aspect is usually challenging for this type of applications, making it a noteworthy advantage of the proposed methodology. The action set of the controller was never constrained or modified, and always had all the heating and cooling actions available. The agent autonomously learned how to blend efficiently heating and cooling in seasonality transitions, while properly focusing only on the important possibilities by itself in more extreme weathers. This suggests that while occasional retraining may be required during the first year of deployment, complete autonomy can be reached with very sparse yearly updates once a complete year of data is obtained or right away from the beginning given a proper reliable model of the building.

The training phase demonstrated an important continuous stability for all instances, with non-degrading and monotonically improving performance. Even after 80% of the epochs completed, pursuing training with a fixed exploration rate of $\epsilon = 0.05$ did not change significantly the actual approximated optimal Q-values following new neural network updates. Such behavior is important to monitor, as it proves convergence toward a stable solution, and can even be used to halt training after the optimal number of training iterations for deployment.

Our general synthesis is that multi-system DQN controller deployment value increases with the complexity and opportunity potential of the considered instance. That is, systems exhibiting complex dynamics and transient behaviors which typically induce challenges to building operators and engineers, like fluctuating energy prices or contracts, a high number of simultaneous control components, dynamic building schedules, or user-defined set points, to name but a few. The *learning* and adaptive capability of DRL methodology then becomes highly profitable, and does not suffer from obsolescence in the long term. Applying this solution to smaller-scale buildings like houses or simpler systems still result in improvements, but can prove more impactful in higher volumes and with group coordination.

## V. Conclusion

Case studies results demonstrate the successful application of our proposed methodology on the smart building multi-objective optimization problem. Significant improvements were observed in expenses, thermal comfort, energy

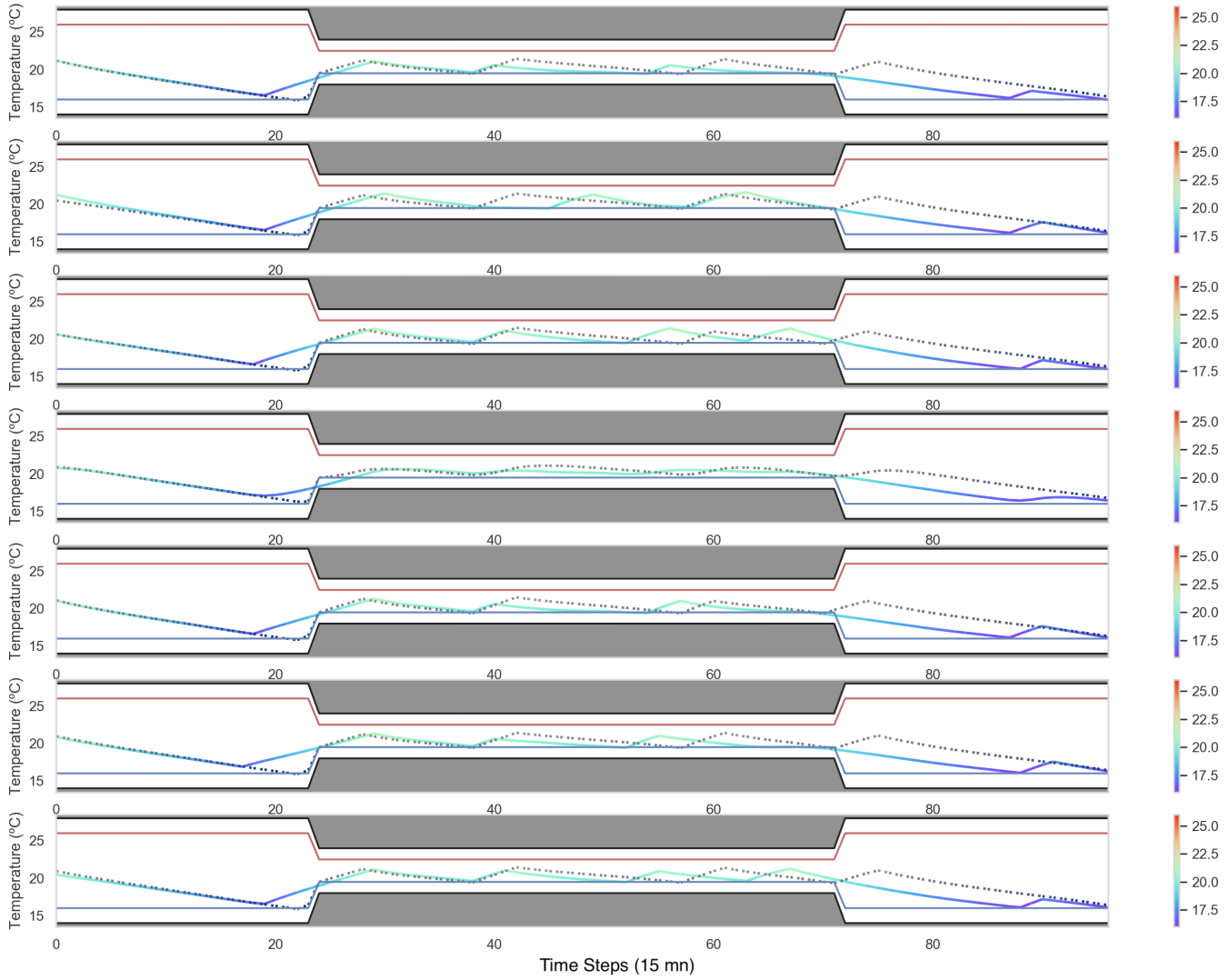


Fig. 9. Temperature variation in each zone for the small commercial center instance during a typical day in January 2020. The fully colored line represents the RL agent's result, while the grey dashed line depicts what a classical controller would have performed under the same conditions.

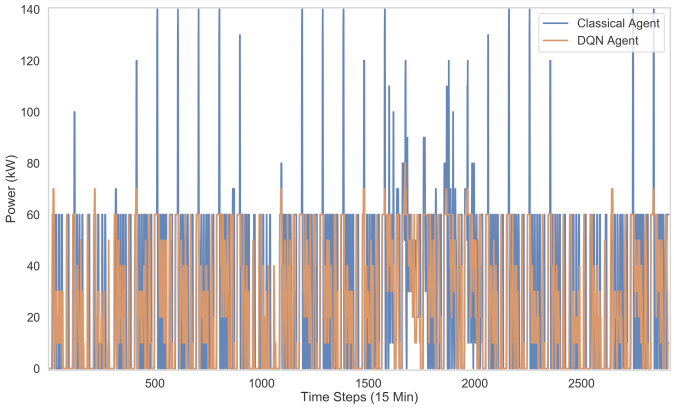


Fig. 10. Instantaneous 15 minutes power calls for phase 1 of instance C.

consumption, power peaks and equipment cycling for different buildings accounting multiple HVAC systems. Our proposed DQN controller outperformed its classical reactive counterpart with respect to all individual objectives in simulation, and showed successful year-round deployment without any retraining.

In addition to its autonomous learning capabilities directly from raw observations of any entity, DRL reward definition can easily be tuned and adapted through a simple graphical user interface (GUI) post-implementation to reflect different optimization objectives. Furthermore, from a calculation perspective, a DQN controller presents the advantage of being light in deployment, as only matrix multiplications are required during the forward pass to access the policy. This can further be leveraged in the GUI to offer visualization and simulation tools as a transparency measure to building managers and operators.

In the light of our results, it is our belief that DRL and its extensions can cope with much more complex smart building instances, both in terms of scale and dynamics. Moreover, generalization success on unseen data leads us to the premise that our approach could be directly applied on a real building in future work, by first performing the learning phase on a

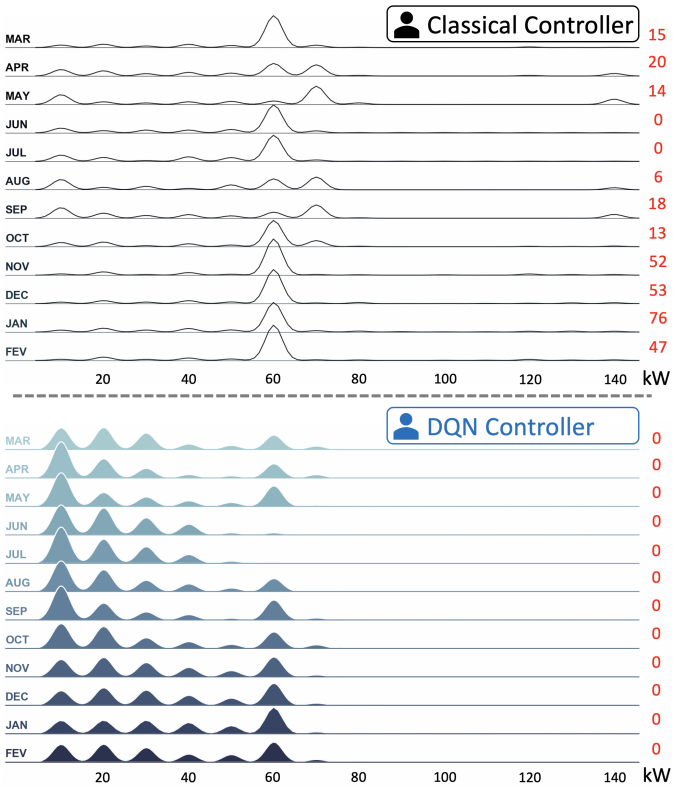


Fig. 11. Monthly non-zero power call distributions for instance C, from March 15 2019 to March 15 2020. The red numbers on the right denote the number of power calls over 120 kW.

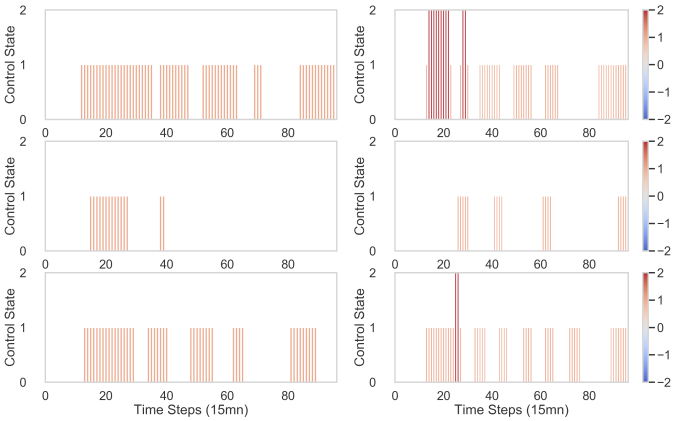


Fig. 12. 24-hour control sequences on instance B for the DQN controller (left) and classical controller (right) during a typical day of January 2020.

digital twin with an appropriate statistical or physical model. Given multiple adoptions, this building-scale solution could then be extended to a larger multi-agent hierarchy at the grid level to help electrical operators and support network resiliency.

## Appendix A
## Deep Learning

Artificial neural networks (ANN) are computing systems structured as alternating layers of aggregated parameters and nonlinear activation functions. Their strength comes from their ability to learn complex hierarchical non-linear representations of different abstraction levels from data samples and to scale particularly well with massive datasets. Traditional fully-connected layers can be seen as vector-to-vector mappings, where two operations are successively applied on a given input $x$: first, a linear transformation of the form

$$Wx + b \ , \tag{7}$$

where $W$ represents the *weights* matrix, and $b$ the *bias* vector; then, the linearly modified outputs are fed as an argument into a non-linear *activation function* $\sigma$. Typical choices of functions are the *rectified linear unit* (ReLU) $\equiv \max(0, x)$, the *hyperpolic tangent* $\equiv \tanh(x)$ or other *sigmoid* functions. The *forward pass* or *forward propagation* of a neural network is defined as the complete information flow from the input layer to the output [20]. For the multi-layer perceptron (MLP), one of the most standard ANN made of consecutive fully connected layers, this yields:

$$h^{(i)} = \sigma^{(i)}\left(W^{(i)\mathrm{T}} h^{(i-1)} + b^{(i)}\right) \ \forall \ i \in [1, 2, ..., L+1] \ , \tag{8}$$

where $h^{(i)}$ is the output of the $i$-th layer, $h^{(0)} = x$, and $L$ is the number of hidden layers. The output $h^{(L+1)} = \hat{y}$ typically uses a different activation function, depending on the nature of the considered supervised learning task.

Deriving a scalar cost $\mathcal{L}(\theta)$ from $\hat{y}$ with $\theta = [W^{(1)} \dots W^{(L+1)}; b^{(1)} \dots b^{(L+1)}]$, and applying the maximum likelihood principle on $n$ available samples, the standard ANN training problem has the form

$$\min_{\theta} \sum_{k=1}^{n} \left(\mathcal{L}\big(\hat{y}_{(k)}(x_{(k)}; \theta), y_{(k)}\big)\right) \ , \tag{9}$$

where $y$ denotes the real value of some predicted quantity. From this point, the gradient of the loss with respect to each parameter $\theta$ in the network can be computed, in a phase referred to as *back-propagation*. Finally, given the unconstrained nonconvex differentiable optimization nature of the problem, such instances are usually addressed with standard (incremental) gradient-type optimization methods [21].

### A. Recurrent Neural Networks

Recurrent neural networks (RNN)s [22] are designed to leverage sequential data of the form $x_1, ..., x_\tau$, by building an internal state $h$ updated at each sequence input $x_t$ by the recursive relationship

$$h_t = f(h_{t-1}, x_t; \theta) \ . \tag{10}$$

This hidden state is used as a partial summary of the task-relevant aspects of past sequence inputs up to time $t$, and can be used in different fashions, depending on how the computational graph of the outputs is developed by the user. Bidirectional RNNs [23] extend classical RNNs by combining two layers of opposite direction to the same output.

Gated RNNs such as the Long Short-Term Memory [24] (LSTM) or the Gated Recurrent Unit [25] (GRU) build on top of classical RNNs by adding internal memory cells and operations to control information flow through different logical gates implemented with sigmoids. More concretely, LSTMs

possess a *forget* gate to control information kept from previous cell states; an *input gate* to filter new information input from $x_t$ and $h_{t-1}$; and an *output* gate to control the nature of the next hidden state $h_{t+1}$. Besides being popular for solving calculation issues known as *exploding* and *vanishing* gradient, internal gated recurrent unit components are also very useful because their behavior and properties can be learned along with the rest of the parameters during the training process.

## APPENDIX B
## THERMODYNAMICS AND MODELING

Starting from the diffusion term in the general physics *transport equation* of some quantity $u$, temperature in our context :

$$\frac{\partial u}{\partial t} = \nabla \cdot (D \nabla u) \ , \tag{11}$$

where $D$ is the diffusion coefficient which we assume constant, we define a spatial meshing of the form $x_j = j \times \Delta x, \ j = 0, 1, ..., J-1$ and temporal meshing $t^n = n \times \Delta t, \ n = 0, 1, ..., N$. The time iteration index $n$ goes up to $N$ because $n = 0$ is used to denote the initial condition.

Given the initial and boundary conditions, converting the left-hand side of (11) into a finite difference for the temporal derivate, and the right side into a second-order centered temporal difference for the second derivative of $x$ leads to the well known *explicit* or *Forward-in-Time-Centered-in-Space* (FTCS) algorithm [26]. Evaluating the right-hand term of the FTCS method at time $n+1$ instead of $n$ results in the so-called *Implicit Euler* method:

$$u_j^{n+1} = u_j^n + \Big(\frac{D\Delta t}{(\Delta x)^2}\Big)(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) \ , \tag{12}$$

that can be expressed under the matrix form

$$K_{ij} \ u_j^{n+1} = u_i^n \ . \tag{13}$$

This method is particularly well suited for applications with bigger disparities between characteristic times, which is the case in thermal modeling of a building.

Focusing on the unidimensional implicit finite-elements heat transfer implementation, we consider that each given node $i$ is exchanging heat with all neighbouring nodes $j$ and $k$ (the latter indexes nodes with known defined temperature, a boundary condition) through conduction, convection and radiation, expressed as an equivalent conductance $U$; has capacitance or thermal mass $C$ = $mc_p$ (J/K), where $c_p$ (J/kg K) is the specific heat of the medium and $m$ (kg) its mass; and is receiving heat from a source $Q$. The finite difference equation can then be written:

$$\sum_j U_{ij}^{t+1}(T_j^{t+1} - T_i^{t+1}) + \sum_k U_{ik}^{t+1}(T_k^{t+1} - T_i^{t+1}) - \frac{C_i}{\Delta T}(T_i^{t+1} - T_i^t) + \dot{Q}_i^{t+1} = 0 \ , \tag{14}$$

where $U_{ij}$ (W/K) is the conductance between nodes $i$ and $j$, $\dot{Q}$ (W) the heat flow into the node and $\Delta t$ is expressed in seconds. The conductance for the conduction, convection and radiation heat transfer mechanisms is respectively given by $Ak/L$, $\bar{h}_c A$ and $\bar{h}_r A$, where $A$ ($m^2$) is the area through which heat is transferred, $k$ (W/m K) the thermal conductivity, $L$ (m) the length between points $i$ and $j$, and $\bar{h}_c$ (W/K $m^2$) and $\bar{h}_r$ (W/K $m^2$) the convection and radiation heat transfer coefficients.

## REFERENCES


[1] International Energy Agency. Energy efficiency: Buildings. https://www.iea.org/topics/energyefficiency/buildings/, 2019. Accessed: 2020-03-08.

[2] Environmental and Energy Study Institute. Buildings and built infrastructure. https://www.eesi.org/topics/built-infrastructure/description. Accessed: 2020-04-1.

[3] UN Environment. Global status report: towards a zero-emission, efficient, and resilient buildings and construction sector. https://www.worldgbc.org/sites/default/files/UNEP%20188_GABC_en%20%28web%29.pdf, 2017. Accessed: 2020-03-15.

[4] D. Lee, S. Lee, P. Karava, and J. Hu. Approximate dynamic programming for building control problems with occupant interactions. In *2018 Annual American Control Conference (ACC)*, pages 3945–3950, 2018.

[5] Berenger Favre and Bruno Peuportier. Optimization of building control strategies using dynamic programming. 08 2013.

[6] Model predictive control for smart buildings to provide the demand side flexibility in the multi-carrier energy context: Current status, pros and cons, feasibility and barriers. *Energy Procedia*, 158:3026 – 3031, 2019. Innovative Solutions for Energy Transitions.

[7] I. Szilagyi and P. Wira. An intelligent system for smart buildings using machine learning and semantic technologies: A hybrid data-knowledge approach. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 20–25, 2018.

[8] Djamel Djenouri, Roufaida Laidi, Youcef Djenouri, and Ilangko Balasingham. Machine learning for smart building applications: Review and taxonomy. *ACM Computing Surveys*, 52, 02 2019.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction (2nd Edition)*. MIT Press, 2018.

[10] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control - Approximate Dynamic Programming - Volume II (4th Edition)*. Athena Scientific, 2005.

[11] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Interscience, 2005.

[12] Christopher Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[13] Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Human-level control through deep reinforcement learning. *Nature*, 518:529–540, 2015.

[14] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *AAAI*, page 2094–2100, 2016.

[15] The Dark Sky Company (2020). Dark sky api. https://darksky.net/dev, 2020. Accessed: 2020-03-18.

[16] Laura Graesser and Wah Loon Keng. *Foundations of Deep Reinforcement Learning - Theory and Practice in Python*. Addison Wesley, 2019.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ArXiV*, 2014.

[18] Refrigerating American Society of Heating and Air-Conditioning Engineers (ASHRAE). *ASHRAE Handbook - Fundamentals*. Atlanta, GA, 2017.

[19] Donald R. Wulfinghoff. *Energy Efficiency Manual*. Energy Institute Press, 1999.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[21] Dimitri P. Bertsekas. *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019.

[22] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by backpropagating errors. *Nature*, 323:533–536, 1986.

[23] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *IEEE*, 45:2673–2681, 1997.

[24] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[25] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[26] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. Cambridge University Press, USA, 1992.

# References


[1] The Dark Sky Company (2020) : Dark sky api. `https://darksky.net/dev`, 2020. Accessed: 2020-03-18.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu et Xiaoqiang Zheng : TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] U.S. Energy Information Administration : International energy outlook 2016. `https://www.eia.gov/outlooks/ieo/pdf/buildings.pdf`, 2016. Accessed: 2020-03-10.

[4] Md Zahangir Alom, Tarek M. Taha, Chris Yakopcic et Stefan Westberg : A state-of-the-art survey on deep learning theory and architectures. 2019. Accessed: 2020-03-20.

[5] Refrigerating American Society of Heating et Air-Conditioning Engineers (ASHRAE) : *ASHRAE Handbook - Fundamentals*. Atlanta, GA, 2017.

[6] Dzmitry Bahdanau, Kyunghyun Cho et Yoshua Bengio : Neural machine translation by jointly learning to align and translate, 2014.

[7] Ian H. Bell, Jorrit Wronski, Sylvain Quoilin et Vincent Lemort : Pure and pseudo-pure fluid thermophysical property evaluation and the open-source thermophysical property library coolprop. *Industrial & Engineering Chemistry Research*, 53(6):2498–2508, 2014.

[8] Yoshua Bengio, Pascal Lamblin, Dan Popovici et Hugo Larochelle : Greedy layer-wise training of deep networks. *NIPS*, 2007.

[9] Dimitri P. Bertsekas : *Dynamic Programming and Optimal Control - Approximate Dynamic Programming - Volume II (4th Edition)*. Athena Scientific, 2005.

[10] Dimitri P. Bertsekas : *Nonlinear Programming, 3rd Edition*. Athena Scientific, 2016.

[11] Dimitri P. Bertsekas : *Dynamic Programming and Optimal Control - Volume I (4th Edition)*. Athena Scientific, 2017.

[12] Dimitri P. Bertsekas : *Reinforcement Learning and Optimal Control*. Athena Scientific, 2019.

[13] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado et Wei Chai : Wide and deep learning for recommender systems. *Deep Learning for Recommender Systems*, pages 7–10, 2016.

[14] Kyunghyun Cho, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk et Yoshua Bengio : Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arxXiv*, 3, 2014.

[15] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho et Yoshua Bengio : Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[16] Nadav Cohen : Understanding optimization in deep learning by analyzing trajectories of gradient descent. https://www.offconvex.org/2018/11/07/optimization-beyond-landscape/, 2018. Accessed: 2020-02-15.

[17] Colah : Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. Accessed: 2020-03-24.

[18] Zhicheng Cui et Wenlin Chen : Multi-scale convolutional neural networks for time series classification. 03 2016.

[19] J. Donahue, L. A. Hendricks, M. Rohrbach, S. Venugopalan, S. Guadarrama, K. Saenko et T. Darrell : Long-term recurrent convolutional networks for visual recognition and description. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):677–691, 2017.

[20] Stuart Dreyfus : *Richard Bellman on the Birth of Dynamic Programming*. University of California, Berkeley, 2005.

[21] John Duchi, Elad Hazan et Yoram Singer : Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2021–2159, 2011.

[22] UN Environment : Global status report: towards a zero-emission, efficient, and resilient buildings and construction sector. https://www.worldgbc.org/sites/default/files/UNEP%20188_GABC_en%20%28web%29.pdf, 2017. Accessed: 2020-03-15.

[23] Environmental et Energy Study Institute : Buildings and built infrastructure. https://www.eesi.org/topics/built-infrastructure/description. Accessed: 2020-04-1.

[24] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar et Pierre-Alain Muller : Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33:917–963, 2019.

[25] USEF Foundation : Smart grid task force. https://www.usef.energy/implementations/smart-grids-task-force-eg3/, 2020. Accessed: 2018-15-12.

[26] Xavier Glorot et Yoshua Bengio : Understanding the difficulty of training deep feedforward neural networks. *Proceedings of Machine Learning Research*, 9:249–256, 2010.

[27] Ian Goodfellow, Yoshua Bengio et Aaron Courville : *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[28] Laura Graesser et Wah Loon Keng : *Foundations of Deep Reinforcement Learning - Theory and Practice in Python*. Addison Wesley, 2019.

[29] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel et Sergey Levine : Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *ArXiV*, 2018.

[30] Trevor Hastie, Robert Tibshirani et Jerome Friedman : *The Elements of Statistical Learning - Second Edition*. Springer, 2008.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren et Jian Sun : Deep residual learning for image recognition, 2015.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren et Jian Sun : Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *ArXiV*, 2015.

[33] Geoffrey Hinton, Nitish Srivastava et Kevin Swersky : Neural networks for machine learning. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, 2012. Accessed: 2020-02-24.

[34] Pawan Jain : Complete guide of activation functions. https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044, 2019. Accessed: 2020-03-20.

[35] Gareth James, Daniela Witten, Trevor Hastie et Robert Tibshirani : *An Introduction to Statistical Learning*. Springer, 2017.

[36] Rafal Jozefowicz, Wojciech Zaremba et Ilya Sutskever : An empirical exploration of recurrent network architectures. *Proceedings of Machine Learning Research*, 2015.

[37] Raimi Karim : Attn: Illustrated attention. https://towardsdatascience.com/attn-illustrated-attention-5ec4a
2019. [Online; consulted 10-April-2020].

[38] Diederik P. Kingma et Jimmy Ba : Adam: A method for stochastic optimization. *ArXiV*, 2014.

[39] Kevin J. Lang, Alex H. Waibel et Geoffrey E. Hinton : A time-delay neural network architecture for isolated word recognition. *Neural Network*, 3:23–43, 1990.

[40] Y. LeCun et B. Denker : Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1:541–551, 1989.

[41] Pierre L'Ecuyer : *Stochastic Simulation*. Département d'Informatique et de Recherche Opérationnelle. unpublished yet.

[42] J. H. Lienhard, IV et J. H. Lienhard, V : *A Heat Transfer Textbook*. Phlogiston Press, Cambridge, MA, 5th édition, août 2019. Version 5.00.

[43] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel et Nicolas Heess : Continuous control with deep reinforcement learning. *ArXiV*, 2019.

[44] Minh-Thang Luong, Hieu Pham et Christopher D. Manning : Effective approaches to attention-based neural machine translation. *arXiv,*, 2015.

[45] Raanan Miller : Utility of the future - an mit energy initiative response to an industry in transition. 2016.

[46] T. M. Mitchell : *Machine Learning*. McGraw-Hill, New York, 1997.

[47] Volodymur Mnih, Koray Kavukcuoglu et David Silver : Human-level control through deep reinforcement learning. *Nature*, 518:529–540, 2015.

[48] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza et Alex Graves : Asynchronous methods for deep reinforcement learning. *ArXiV*, 2016.

[49] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho et Yoshua Bengio : On the number of linear regions of deep neural networks. pages 2924–2932, 2014.

[50] Chris Olah et Shan Carter : Attention and augmented recurrent neural networks. https://distill.pub/2016/augmented-rnns/#attentional-interfaces, 2016. [Online; consulted 5-April-2020].

[51] Judea Pearl : *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot et E. Duchesnay : Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[53] Warren Powell : A unified framework for stochastic optimization. *European Journal of Operational Research*, 275, 07 2018.

[54] Warren Powell et Stephan Meisel : Tutorial on stochastic optimization in energy—part i: Modeling and policies. *IEEE Transactions on Power Systems*, 31:1–9, 04 2015.

[55] Warren Powell et Stephan Meisel : Tutorial on stochastic optimization in energy—part ii: An energy storage illustration. *IEEE Transactions on Power Systems*, 31:1–8, 05 2015.

[56] Martin L. Puterman : *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Interscience, 2005.

[57] Herbert Robbins et Sutton Monro : A stochastic approximation method. *Ann. Math. Statist.*, 22(3): 400–407, 09 1951.

[58] David E. Rumelhart, Geoffrey E. Hinton et Ronald J. Williams : Learning representations by back-propagating errors. *Nature*, 323, 1986.

[59] David E. Rumelhart, Geoffrey E. Hinton et Ronald J. Williams : Learning representations by backpropagating errors. *Nature*, 323:533–536, 1986.

[60] Daniel Saunders : The bias-variance tradeoff. `https://djsaunde.wordpress.com/2017/07/17/the-bias-variance-tradeoff/`, 2017. Accessed: 2020-03-10.

[61] John Schulman, Sergey Levine, Philipp Moritz et Michal Jordan : Trust region policy optimization. *ArXiV*, 2015.

[62] John Schulman, Filip Wolski, Prafulla Dhariwal et Alec Radford : Proximal policy optimization algorithms. *ArXiV*, 2017.

[63] M. Schuster et K.K. Paliwal : Bidirectional recurrent neural networks. *IEEE*, 45:2673–2681, 1997.

[64] scikit-optimize contributors : scikit-optimize documentation. `https://scikit-optimize.github.io/0.7//_downloads/scikit-optimize-docs.pdf`, 2020. Accessed: 2020-02-28.

[65] David Silver, Aja Huang J. Maddison, Arthur Guez et Laurent Sifre : Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[66] David Silver, Julian Schrittwieser et Karen Simonyan : Mastering the game of go without human knowledge. *Nature*, 550:354–375, 2017.

[67] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever et Ruslan Salakhutdinov : Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[68] Maxwell Stinchcombe et Halbert White : Multilayer feedforward networks are universal approximators. *University of California*, 2:359–366, 1989.

[69] Ilya Sutskever, James Martens, George Dahl et Geoffrey Hinton : On the importance of initialization and momentum in deep learning. *ArXiV*, 2013.

[70] Ilya Sutskever, Oriol Vinyals et Quoc V. Le : Sequence to sequence learning with neural networks. *International Conference on Neural Information Processing Systems*, 2:3104–3112, 2014.

[71] Ilya Sutskever, Oriol Vinyals et Quoc V.Le : Sequence to sequence learning with neural networks. *arxXiv*, 3, 2014.

[72] Richard S. Sutton et Andrew G. Barto : *Reinforcement Learning - An Introduction (2nd Edition)*. MIT Press, 2018.

[73] Richard S. Sutton, David McAllester, Satinder Singh et Yishay Mansour : Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing systems*, 12:1057–1063, 2000.

[74] Johnson Thomas : Latest trends in smart grid technology in the utilities industry. `https://medium.com/@mikethomsan/latest-trends-in-smart-grid-technology-in-the-utilities-industry-9e2f295d3a4f/`, 2019. Accessed: 2019-11-15.

[75] Hado van Hasselt : Double q-learning. *Advances in Neural Information Processing Systems*, pages 229–256, 2010.

[76] Hado van Hasselt, Arthur Guez et David Silver : Deep reinforcement learning with double q-learning. *ArXiV*, 2015.

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Gomez Aidan N, Lukasz Kaiser et Illia Polosukhin : Attention is all you need. *Neural Information Processing Systems*, pages 7–10, 2017.

[78] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano et Kevin J. Lang : Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37, 1989.

[79] Ziyu Wang, Nando de Freitas et Marc Lanctot : Dueling network architectures for deep reinforcement learning. *ArXiV*, 2015.

[80] Christopher Watkins et Peter Dayan : Q-learning. *Machine Learning*, 8:279–292, 1992.

[81] Ronald J. Williams : Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[82] D. Randall Wilson et Tony R. Martinez : The general inefficiency of batch training for gradient descent learning. *Elsevier - Neural Networks*, 16:1429–1451, 2003.

[83] Donald R. Wulfinghoff : *Energy Efficiency Manual*. Energy Institute Press, 1999.

[84] Ke Ye et Lek-Heng Lim : Every matrix is a product of toeplitz matrices. *ArXiV*, 2013.

[85] Yi Tao Zhou et Rama Chellappa : Computation of optical flow using a neural network. *MDPI electronics*, 8, 2019.